Kimmo Myllymäki

SBTS-TUKIASEMAN TOIMINNALLISTEN TILOJEN TARKISTAMINEN TESTAUSAUTOMAATIOLLA

SBTS-TUKIASEMAN TOIMINNALLISTEN TILOJEN TARKISTAMINEN TESTAUSAUTOMAATIOLLA

Kimmo Myllymäki
Opinnäytetyö
Syksy 2016
Teknologialiiketoiminta, Ylempi AMK
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Master-tutkinto, Teknologialiiketoiminta

Tekijä(t): Kimmo Myllymäki
Opinnäytetyön nimi: SBTS-tukiaseman toiminnallisten tilojen tarkistaminen testausautomaatiolla
Työn ohjaaja: Juha Alakärppä
Työn valmistumislukukausi ja -vuosi: Syksy 2016 Sivumäärä: 74

Mobiiliverkoissa välitettävä liikenne kasvaa edelleen ja verkko-operaattoreilla onkin tarve laajentaa erityisesti LTE-radioverkkoja olemassa olevien GSM- ja WCDMA -tekniikoiden rinnalle. Erillisen LTE-tukiaseman rakentamisen sijaan markkinoilla on tarve arkkitehtuuriltaan yksinkertaiselle, ohjelmallisesti muunneltavalle ja useampaa radioteknologiaa yhtä aikaa tukevalle tukiasematuotteelle ja Nokian uusi SRAN/SBTS-tukiasema on eräs vaihtoehto. Useasta yhtäaikaisesta radiotekniikasta, kasvavasta liikenteestä ja tukiasemien suuresta määrästä johtuen myös verkon hallinta asettaa verkko-operaattoreille kasvavia haasteita. Nokian tarjoama ratkaisu tähän on Nokia NetAct verkonhallintajärjestelmä.

Ohjelmistojen testaus on oleellinen osa tuotekehitystä. Testausta voidaan tehdä sekä manuaalisesti että automaattisesti. Työn tavoitteena oli kehittää testitapauksia erilaisilla teknisillä toteutustavoilla valitussa testausautomaatiojärjestelmässä, vertailla erilaisia toteutuksia keskenään ja arvioida testausautomaatiolla saavutettavaa hyötyä manuaaliseen testaukseen verrattuna. Työ tehtiin Nokia Networks / Mobile Networks Oulun yksikössä.

Opinnäytetyössä käytettiin aineistona aiheeseen liittyvää kirjallisuutta ja internetistä löytyvää tietoa sekä omaa aiempaa kokemusta tukiasemien tuotannossa, asiakastukitehtävissä ja manuaalisessa testauksessa.

Työssä kehitettiin seitsemän testitapausta käyttäen erilaisia teknisiä toteutustapoja, joissa tarkistetaan tukiaseman toiminnallisia tiloja sekä SRAN/SBTS-tukiaseman että NetAct:n kautta. Tulosten perusteella testattavan ohjelmiston graafista käyttöliittymää hyödyntävät toteutustavat olivat testauskattavuudeltaan parhaita, mutta toisaalta helposti rikkoutuvia. Komentorivipohjaiset toteutukset eivät yltäneet samaan kattavuuteen, mutta olivat toisaalta luotettavimpia ja edullisimpia sekä kehittämisen että ylläpidon taloudellisuuden perusteella.

Kehitetyt testitapaukset voidaan integroida osaksi suurempaa testikokonaisuutta. Näitä ovat muun muassa testattavan tukiaseman toiminnallisten tilojen tarkistus ennen varsinaisen testikokonaisuuden ajoa tai usein toistettavan testiajon sisällä.

Asiasanat:
Mobiiliverkot, ohjelmiston testaus, ohjelmiston uudelleenkäyttö, testausautomaatio, Robot Framework

ABSTRACT

Oulu University of Applied Sciences
Master's degree, Technology Business

Author(s): Kimmo Myllymäki
Title of thesis: Checking of SBTS base station operational states with test automation
Supervisor(s): Juha Alakärppä
Term and year when the thesis was submitted: Autumn 2016 Number of pages: 74

The amount of traffic transferred in mobile networks is ever increasing and the network operators are forced to expand especially the LTE radio network in parallel to existing GSM and WCDMA networks. Instead of building a separate LTE base station there is a market need for a base station which has simplified architecture, is software modifiable and is able to support multiple radio technologies simultaneously. The new SRAN/SBTS base station from Nokia is one solution. Due to multiple simultaneous radio technologies, increasing traffic and huge amount of base stations, network monitoring is also a challenge for network operators. Nokia's solution is Nokia NetAct, a network management system.

Software testing is an important part of research and development. Testing can be performed manually and automatically. The objective of this thesis was to develop test cases by using different technical approaches in a chosen test automation framework, to compare the test cases and to evaluate the benefits achieved by test automation compared to manual testing.

The thesis is based on literature, information available in the internet and my own experiences in base station production, customer support, research and development and manual testing.

Seven different test cases were developed by using different technical approaches, which check the base station operational states via SRAN/SBTS and NetAct. Based on the results, the test cases based on graphical user interface were the most comprehensive but they could easily break. Test cases based on command line interface were not as comprehensive but had better reliability and were also more economical from development and maintenance cost point of view.

The developed test cases can be integrated as part of a bigger test entity. For example, the test cases can be used to check the operational states of the base station before the test run is started. Furthermore, the test cases can be used as a part of continuous testing.

Keywords:
Mobile networks, software testing, software reuse, test automation, Robot Framework

ALKULAUSE

Haluan ensinnäkin kiittää esimiestäni Juho Pesosta mahdollisuudesta tehdä tämä opinnäytetyö mielenkiintoisesta ja itselleni uudesta aiheesta. Samalla haluan kiittää työni ohjaajaa Hannu Kaarelaa saamastani ohjauksesta, hyvistä neuvoista ja positiivisesta suhtautumisesta opinnäytetyöhöni. Erityisen suuret kiitokset kuuluvat Aarno Parkkilalle, Lauri Pajuselle, Vesa Hyvöselle, Kari Saukolle, Jari Karhulalle ja Hannu Koskelle lukemattomista vinkeistä, kommenteista, rakentavista keskusteluista ja kärsivällisyydestä kysyessäni samoja asioita joskus useammankin kerran.

Oulun ammattikorkeakoululta haluan kiittää työni ohjaajaa Juha Alakärppää rakentavista kommenteista ja kannustavasta suhtautumisesta koko prosessin ajan. Muista työkiireistäni johtuen opinnäytetyön alkuperäinen aikataulu venyi hiukan mutta lopulta sain tämän tehtyä.

Rakasta aviovaimoani Päiviä haluan kiittää tuesta ja kannustuksesta koko opiskeluajalta. Suuret kiitokset myös lapsillemme Elinalle, Annille, Aleksille ja Ellalle, koska jaksoitte kestää isän tietokoneen ääressä vietettyjä lukemattomia iltoja ja viikonloppuja erityisesti tämän opinnäytetyön kirjoittamisen aikana.

Omistan tämän työn perheelleni sillä ilman teitä, teidän tukea ja ymmärrystä opiskelustani ei olisi tullut mitään.

Jäälissä 28.11.2016

SISÄLLYS

SANASTO

3GPP	Standardointijärjestöjen yhteistyöorganisaatio (3rd Generation Partnership Project)
BSC	Tukiasemaohjain (Base Station Controller)
BIM	Tukiaseman informaatiomalli (Base station Information model)
CLI	Komentorivi (Command Line Interface)
CRT	Jatkuva regressiotestaus (Continuous Regression Test)
DCN	Datasiirtoverkko (Data Communication Network)
HTML	Verkkosivujen kuvauskieli (Hypertext Markup Language)
e2e	Päästä päähän -testaus (end-to-end)
Firebug	Firefox -verkkoselaimen lisäosa verkkosivun teknisen sisällön tutkimiseen
Firewall	Ohjelmisto tai laite kahden tietoverkon välisen liikenteen suodattamiseen
Flame	Nokian sisäinen testaustyökalu
Java	Ohjelmointikieli
Jenkins	Automaatiopalvelin
JRE	Java-kielen ajoympäristö (Java Runtime Environment)
JSpy	Java-sovellusten kehitystyökalu
Keyword	Avainsana
MACG	Puhelugeneraattori (Massive Android Call Generator)
MCG	Puhelugeneraattori (Mobile Call Generator)
MML	Tukiasemaohjaimissa käytettävä komentokieli (Man Machine Language)
MO	Hallittava olio (Managed Object)
OSS	Verkon hallintajärjestelmä (Operations Support System)
PLMN	Yleinen matkapuhelinverkko (Public Land Mobile Network)
Python	Ohjelmointikieli
RAT	Radioteknologia (Radio Access Technology)
RemoteSwingLibrary	Robot Frameworkin sisältämä kirjasto Java-sovellusten lukemiseen
RNC	Tukiasemaohjain (Radio Network Controller)
Robot Framework	Testausautomaatiotyökalu
SBTS	Single RAN Base Station, Nokian uusi tukiasematyyppi
Selenium2Library	Robot Frameworkin sisältämä kirjasto verkkosivujen lukemiseen

Shell script	Komentorivillä ajettava skripti
Splunk	Työkalu informaation esittämiseen
SRAN	Single Radio Access Network
SVN	Ohjelmistojen versionhallintatyökalu (Subversion)
SyVe	Järjestelmätestaus (System Verification)
Testware	Testausautomaatiossa käytettävä testausohjelmisto
Ubuntu	Eräs Linux-käyttöjärjestelmän versio
UE	Mobiiliverkon käyttäjän päätelaite (User Equipment)
VI	Virtualisoitu infrastruktuuri (Virtual Infrastructure)
VirtualBox	Käyttöjärjestelmien virtualisointiin käytettävä tietokoneohjelma
WebUI	SBTS-tukiaseman verkkoselaimessa toimiva käyttöliittymä
Xpath	Ei-XML-pohjainen kieli XML-dokumenttien osien osoittamiseen
Xubuntu	Eräs Linux-käyttöjärjestelmän versio

1 JOHDANTO

1.1 Taustaa

Neljännen sukupolven LTE-radioverkkojen yleistymisestä huolimatta markkinoilla on edelleen tarve myös toisen sukupolven GSM- ja kolmannen sukupolven WCDMA-radioverkoille. GSM-verkkojen elinkaari on jatkunut jo yli 25 vuotta eikä merkkejä GSM-verkkojen laajamittaisesta alasajosta ole vieläkään. Verkko-operaattoreilla onkin tarve laajentaa LTE-radioverkkoja olemassa olevien GSM- ja WCDMA -tekniikoiden rinnalle. Eräs ratkaisu on asentaa entisen tukiaseman viereen uusi LTE-tukiasema. Ratkaisuna tämä on kuitenkin operaattorille kallis, sillä uusi yksittäistä radiotekniikkaa tukeva tukiasema vaatii omat raudat ja ohjelmistot sekä joissakin tapauksissa myös antennit antennilinjastoineen. Ratkaisu myös lisää verkko-elementtien määrää entisestään tehden verkon hallinnasta yhä haastavampaa. Nokian kehittämä uusin ratkaisu tähän kasvavaan tarpeeseen on SRAN/SBTS (Single RAN Base Station), jossa yksi SRAN/SBTS-tukiasema voi tukea yhtä aikaa yhtä, kahta tai jopa kaikkia kolmea edellä mainittua radiotekniikkaa.

Toisaalta mobiiliverkkojen elementtien lisääntyessä, verkkojen monimutkaistuessa ja liikennemäärien kasvaessa myös verkon hallinta ja liikenteen seuranta sekä tarvittavan kapasiteetin ja investointien hallinta on yhä vaativampaa. Nokian tarjoama ratkaisu tähän tarpeeseen on Nokia NetAct, johon SRAN/SBTS-tukiasema on helppo integroida.

Kaikkiin nykyaikaisiin mobiiliverkkoihin liittyy paljon erilaisia verkkoelementtejä, ohjelmistoja ja toiminnallisuuksia ja järjestelmätasolla tarkasteltaessa mobiiliverkko onkin erittäin monimutkainen kokonaisuus. Tämän opinnäytetyön kannalta keskeisiä tuotteita ja ohjelmistoratkaisuja ovat edellä mainitut Nokia SRAN/SBTS sekä Nokia NetAct.

1.2 Opinnäytetyön rajaus ja tavoitteet

Opinnäytetyössä tutustutaan mobiiliverkkojen markkinanäkymiin, tuoteprosessiin, Nokia NetAct ja SRAN/SBTS-tuotteisiin ja niihin liittyvään verkon rakenteeseen, ohjelmistojen testauksen yleisiin periaatteisiin sekä testausautomaatioon. Työ keskittyy Nokia SRAN/SBTS-ratkaisun järjestelmäta-

son testauksen automatisointiin, siihen soveltuvan testausautomaatiotyökalun valintaan sekä automaatio-ohjelmiston uudelleenkäyttöön. Mobiiliverkko, järjestelmätason testaus ja testausautomaatio ovat aiheina erittäin laajoja. Opinnäytetyö onkin rajattu koskemaan vain tiettyjä työn yhteydessä kehitettyjä ja automatisoituja järjestelmätason testitapauksia.

Opinnäytetyössä kehitetään testausautomaatiolla SBTS-tukiaseman toiminnallisten tilojen tarkistamiseen liittyviä testitapauksia eri menetelmillä. Työssä myös arvioidaan testausautomaation tuomaa hyötyä manuaaliseen testaukseen verrattuna. Työn lopussa pohditaan saavutettuja tuloksia, kehitettyjen testitapausten hyödynnettävyyttä sekä jatkokehitystarpeita.

Opinnäytetyön tavoitteena on:

- Kehittää testitapauksia erilaisilla vaihtoehdoilla valitussa testausautomaatiojärjestelmässä.
- Vertailla vaihtoehtojen ominaisuuksia ja tuloksia keskenään.
- Arvioida testausautomaatiolla saavutettavaa hyötyä manuaaliseen testaukseen verrattuna.

Opinnäytetyössä käytettiin aineistona aiheeseen liittyvää kirjallisuutta ja internetistä löytyvää tietoa sekä omaa aiempaa kokemusta tukiasemien tuotannossa, asiakastukitehtävissä ja manuaalisessa testauksessa. Aikaisempaa kokemusta tukiasemien tai järjestelmätason testausautomaation kehittämisestä tai kehityksessä käytettävistä työkaluista ei juurikaan ollut. Tämän opinnäytetyön tekeminen tarjosikin erinomaisen mahdollisuuden oppia uutta.

SRAN/SBTS-järjestelmätason testausautomaatiota kehitetään tiimityönä projektityyppisesti, joten tämän työn ulkopuolella on toteutettu paljon muitakin testitapauksia ja kehitetty myös testausympäristöjä.

1.3 Yleistä Nokia Networks Oy:stä

Nokia Oyj on suomalainen maailmanlaajuisesti toimiva tietoliikennealan yhtiö. Nokia on viime aikoina muuttanut muotoaan myymällä omia liiketoiminta-alueitaan (kuten esimerkiksi matkapuhelinliiketoiminnan Microsoftille) ja toisaalta ostamalla kilpailevia tai muuten kiinnostavilla alueilla toimivia yrityksiä ja sulauttanut ne omaan toimintaansa. Tällä hetkellä Nokian pääliiketoiminta-alueita ovat verkkoinfrastruktuuriin keskittyvä Nokia Networks sekä teknologiakehitykseen ja lisensointiin

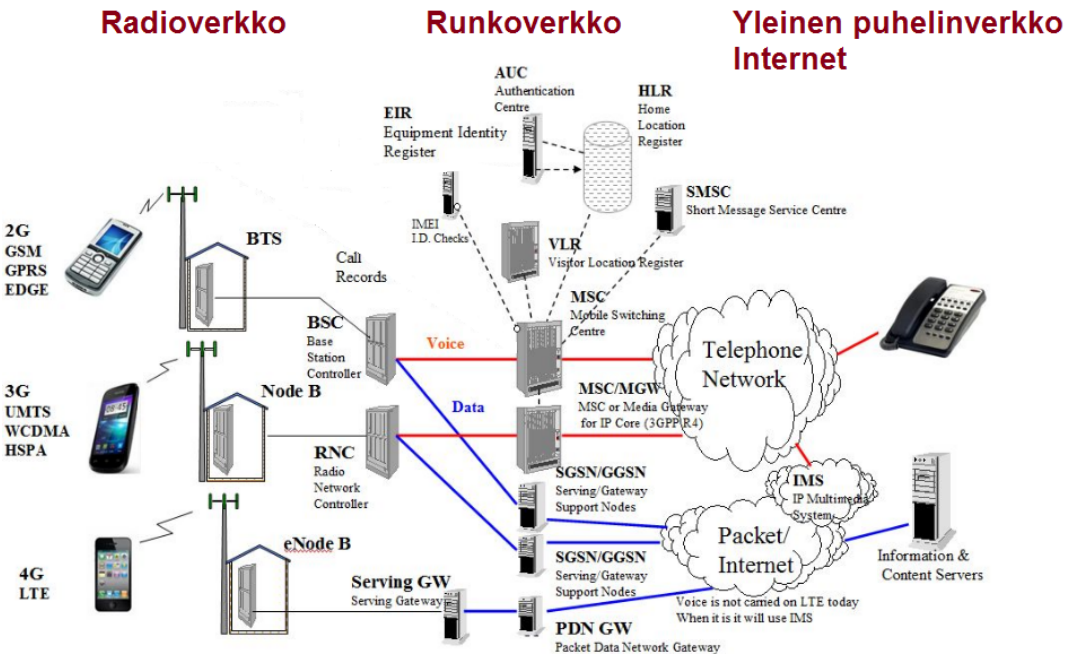
keskittyvä Nokia Technologies. Tässä yhteydessä keskeisin liiketoiminta-alue on verkkoihin keskittyvä Nokia Networks, joka on jakautunut neljään liiketoimintaryhmään. Mobile Networks toimittaa erilaisia langattomia verkkoratkaisuja asiakkaille, Fixed Networks keskittyy langallisten ratkaisujen kehittämiseen ja tarjoamiseen, IP/Optical Networks kehittää IP-reititysteknologioita ja optisia verkkoja sekä Applications & Analytics, joka kehittää ohjelmistoja verkon statistiikan keräämiseen ja analysointiin. (Nokia 2016, viitattu 28.11.2016.)

Tämä opinnäytetyö on tehty Nokia Networks / Mobile Networks Oulun yksikössä, jossa keskitytään radioverkkojen tutkimukseen, tuotekehitykseen ja laitevalmistukseen ja jossa työskentelee tällä hetkellä noin 2 300 henkilöä.

2 MOBIILIVERKOT

2.1 Mobiiliverkon rakenne

Nykyaikaiset mobiiliverkot voidaan jakaa kuvassa 1 olevan yksinkertaistetun kaavion mukaan kahteen pääosaan eli radioverkkoon ja runkoverkkoon. Mobiiliverkko on runkoverkon kautta yhteydessä sekä yleiseen puhelinverkkoon että internetiin. Eri verkkoelementtejä yhdistää siirtoverkko, jota kuvassa 1 esittää yhtenäiset viivat verkko-elementtien välillä.



KUVA 1. Mobiiliverkon rakenne (3gwiz 2016, viitattu 28.11.2016)

Kaikkien radioteknologioiden tukiasemat tarjoavat ilmarajapinnan kautta tilaajien päätelaitteille (esimerkiksi matkapuhelimet) yhteyden radioverkkoon, johon kuuluvat tukiasemien lisäksi kuvan 1 mukaan niitä ohjaavat tukiasemaohjaimet BSC (Base Station Controller) ja RNC (Radio Network Controller). Radioverkon kautta päätelaitteilla on yhteys runkoverkkoon, joka mahdollistaa pääsyn yleiseen puhelinverkkoon sekä internettiin. Runkoverkon elementit huolehtivat myös muun muassa tilaajatiedoista, tilaajien tunnistuksesta ja laskutuksesta.

Radioverkkoon kuuluvien tukiasemien ja niiden ohjaimien tärkein tehtävä on huolehtia ilmarajapinnan kautta radiotaajuuksilla tapahtuvasta kommunikoinnista tukiasemien ja päätelaitteiden (kuten

matkapuhelimien) välillä. Radioverkkojen toiminta perustuu avoimeen radiorajapintaan, yhteisesti ja tarkasti sovittuihin julkisiin standardeihin, määritteisiin sekä muihin rajapintoihin, joita kehittää ja ylläpitää usean eri standardointijärjestön yhteistyöorganisaatio 3GPP (3rd Generation Partnership Project). Yleisimmät maailmanlaajuiset ja 3GPP:n määrittämät radioverkkostandardit tällä hetkellä ovat toisen sukupolven GSM- (Global System for Mobile Communications), kolmannen sukupolven WCDMA- (Wideband Code Division Multiple Access) ja neljännen sukupolven LTE-verkot (Long Term Evolution). Viidennen sukupolven eli 5G-verkkojen kehitys on jo vauhdissa mutta tätä kirjoitettaessa 5G-standardointityö on vielä kesken.

GSM-järjestelmästä käytetään usein lyhennettä 2G (2nd Generation), WCDMA:sta 3G (3rd Generation) ja LTE:stä 4G (4th Generation). Käytettävä lyhenne riippuu usein asiayhteydestä, ja myös tässä opinnäytetyössä esiintyy molempia nimeämistapoja. Radioteknologiasta käytetään yleisesti myös lyhennettä RAT (Radio Access Technology) ja molempia nimitystapoja esiintyy myös tämän työn yhteydessä.

2.2 Radioverkkojen markkinanäkymät

Mobiiliverkkojen kokonaistilaajamäärä jatkaa edelleen kasvuaan ja maailmanlaajuinen mobiililiittymien määrä oli yli 7,6 miljardia vuoden 2015 lopussa, joka vastaa 4,7 miljardia uniikkia tilaajaa. Kasvu painottuu älypuhelimissa ja muissa laitteissa käytettäviin laajakaistaliittymiin ja onkin arvioitu, että niiden osuus kaikista liittymistä saavuttaa 71% vuonna 2020 ollen 47% vuonna 2015. Laajakaistaliittymien myötä verkoissa siirrettävän datan määrän on arvioitu kasvavan 49% vuodessa seuraavien 5 vuoden aikana. Datan siirtoon käytettävien laajakaistaliittymien asettamista vaatimuksista johtuen kasvu painottuu erityisesti 3G- ja 4G-radioverkkoihin. Niiden kasvusta huolimatta 2G-radioverkot hallitsevat edelleen, vaikkakin lasku on jo alkanut. Eräänä mielenkiintoisena kasvualueena mainitaan myös M2M- (Machine-to-Machine) markkinat. (GSMA 2016, 6–7. The Mobile Economy 2016, viitattu 28.11.2016.)

Liikenteen kasvusta johtuen verkko-operaattorit joutuvatkin tekemään tarkkoja suunnitelmia operaattorin toimiluvassa määriteltyjen, rajallisten taajuusalueiden käytöstä eri verkkoteknologioiden välillä. Ainakin lähivuosien ajan sekä laitevalmistajien että verkko-operaattoreiden pitää tukea ja ylläpitää niin 2G-, 3G- kuin LTE-teknologioita niiden erilaisista sovellustarpeista johtuen.

Radioverkkojen tukiasemat olivat pitkään fyysiseltä kooltaan isoja sekä olemukseltaan kaappimaisia. Niiden sisällä oli paikat tarvittaville teholähteille sekä digitaalisille ja radiotaajuisille pistoyksiköille. Tukiaseman asennus, sisäinen kaapelointi ja konfigurointi suoritettiin käyttöönoton aikana. Monia vaiheita sisältänyt prosessi edellyttikin asennushenkilöstöltä erityistä ammattitaitoa. Lopputuloksena syntyi yhden radiotekniikan eli tyypillisesti joko 2G- (GSM) tai 3G- (WCDMA) tukiasema, jonka muuttaminen jälkikäteen etäyhteydellä oli usein vaikeaa tai jopa mahdotonta. Tyypillisesti tällainen tukiasema oli myös fyysisesti iso. Se asennettiin joko sisätiloihin tai vaikkapa talon katolle ulos. Tukiaseman tehonkulutus ja sen vuoksi lämmöntuotto oli suuri vaatien joissakin kohteissa erillisen jäähdytyskoneiston toimiakseen luotettavasti. Markkinoilla onkin tarve arkkitehtuuriltaan yksinkertaisemmalle ja useampaa radioteknologiaa yhtä aikaa tukevalle tukiasematuotteelle ja Nokian uusi SRAN/SBTS-ratkaisu on eräs vaihtoehto.

Mobiiliverkon liikenteen muuttuessa verkko-operaattorit joutuvat laajentamaan runkoverkon kapasiteettia, päivittämään verkko-elementtien ohjelmistoja sekä tekemään uudistuksia erityisesti tukiasemien laitteistoissa ja konfiguraatioissa, Kaikki tämä näkyy kasvavana testaustarpeena myös Nokian tuotekehityksessä.

2.3 Mobiiliverkon hallinta

Mobiiliverkoissa välitettävän liikenteen lisääntyessä ja muuttaessa jatkuvasti muotoaan on operaattoreiden jatkuvasti tarkkailtava verkkojaan ja suunniteltava muutoksia ja laajennuksia etukäteen. Esimerkiksi Suomen operaattoreiden mobiiliverkoissa välitettävän datan määrä on kaksinkertaistunut useina vuosina peräkkäin, eikä kasvuvauhti osoita hiipumisen merkkejä (Yle 2016, viitattu 28.11.2016).

Operaattoreiden onkin jatkuvasti seurattava verkon kuormitusta ja suunniteltava ja toteutettava verkon laajennuksia siten, että operaattorin toimiluvassa määritellyt ehdot ja toisaalta loppukäyttäjien odotukset täyttyvät. Kaikki tämä edellyttää koko mobiiliverkon kattavaa ja keskitettyä hallintaa, jotta verkon liikenteen seurannasta kertyvästä valtavasta tietomäärästä voidaan tehdä analyysejä ja etsiä verkon pullonkauloja. Yhä monimutkaistuvien verkkojen monitorointi ja hallinta onkin eräs verkko-operaattoreiden tärkeimmistä tehtävistä ja toisaalta suurimmista haasteista.

3 TUOTEPROSESSI

Yleisesti ottaen liiketoiminnassa tuoteprosessin voidaan nähdä tarkoittavan koko yritystä koskevaa kokonaisvaltaista ajattelua, jonka lähtökohtana on yrityksen perusasian eli kannattavan tuotteen aikaan saaminen (Päätalo 2015, 3).

Nokia Networksin tuoteprosessi on hyvin laaja ja siihen kuuluu lukuisia erillisiä dokumentteja. Prosessi on esitelty sisäisessä Nokia Create Process - Introduction –dokumentissa. Dokumentti kuvaa prosessin kannalta keskeiset käsitteet, itse prosessin, siihen liittyvän päätöksenteon ja komponentit. Esiteltyihin käsitteisiin kuuluvat muun muassa tuote (product), sovellus (application), verkkokokonaisuus (network entity), alusta (platform), järjestelmä (system), julkaisu (release), ohjelma (program), projekti (project) ja ominaisuus (feature). (Nokia Create Process – Introduction 2016, 6-8.)

Tämän opinnäytetyön kannalta keskeisiä Nokia Networksin sisäisen tuoteprosessin tuloksena syntyneitä ja jo markkinoilla olevia tuotteita ovat erityisesti SRAN/SBTS sekä NetAct. Olennainen osa Nokia Networksin tuoteprosessia ovat ohjelmistojen kehitys ja niiden testaus. Ne muodostavat perustan myös tälle opinnäytetyölle.

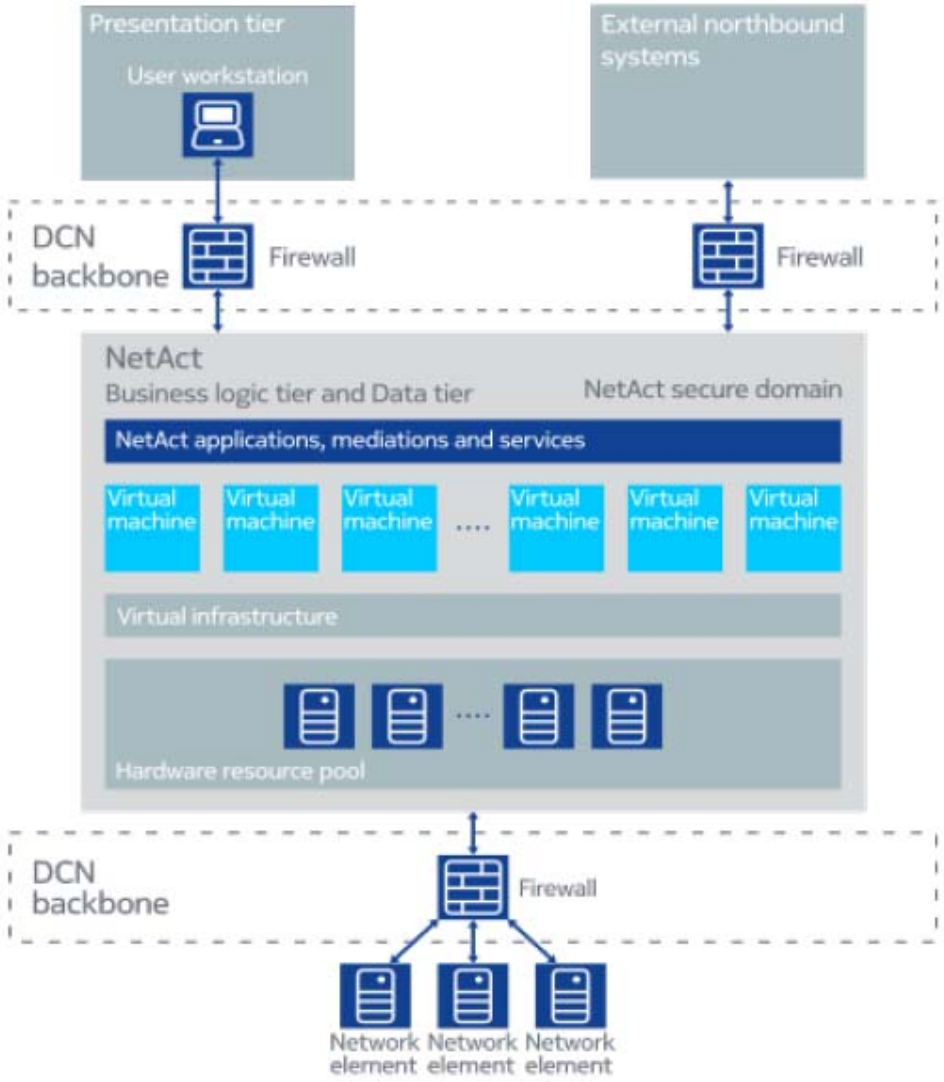
4 NOKIA NETACT

4.1 Nokia NetActin kuvaus

Nokia NetAct on ohjelmistoratkaisu, johon kuuluu monia toimintoja ja sovelluksia kokonaisen mobiiliverkon ja sen sisältämien radio- ja runkoverkon elementtien hallintaan. Verkon hallintatyökaluista käytetään yleisellä tasolla usein myös lyhennettä OSS (Operations Support Systems). Nokia NetActin avulla operaattori voi yhdestä paikasta seurata ja hallita sekä koko mobiiliverkkoa että sen yksittäistä verkkoelementtiä, vaikkapa yksittäistä tukiasemaa. (Nokia NetAct 2016, viitattu 28.11.2016.)

NetAct sisältää useita eri toiminnallisia lohkoja ja niiden sisällä olevia sovelluksia, jotka toimivat niin sanottujen virtuaalikoneiden eli VM:ien (Virtual Machine) sisällä. NetAct liittyy mobiiliverkon verkkoelementteihin eteläisten rajapintojen (Southbound Interfaces) kautta ja toisaalta käyttäjän työasemiin sekä ulkopuolisiin tietojärjestelmiin pohjoisten rajapintojen (Northbound Interfaces) kautta. Liikenne NetActin ja ulkoisten elementtien välillä kulkee datasiirtoverkon eli DCN:n (Data Communication Network) kautta, joissa käytetään palomuureja (firewalls) suojaamaan verkon sisällä kulkevaa liikennettä ulkopuolelta tulevia hyökkäyksiä vastaan. NetActin ohjelmisto pyörii virtualisoidun infrastruktuurin eli VI:n (Virtual Infrastructure) päällä, jolloin asiakkaat voivat käyttää sitä erilaisissa laitteistoissa ja käyttöjärjestelmissä. Laitteiston tarjoamat yhteiset resurssit muodostavat resurssipoolin, joita jaetaan pienempiin yksiköihin eli edellä mainittuihin virtuaalisiin koneisiin. (Nokia NetAct 16.2 System Overview 2016, 4-34.)

NetActin arkkitehtuuria ja ulkoisia rajapintoja esitetään kuvassa 2.



KUVA 2. NetActin arkkitehtuuri ja ulkoiset rajapinnat (Nokia NetAct 16.2 System Overview 2016, 20)

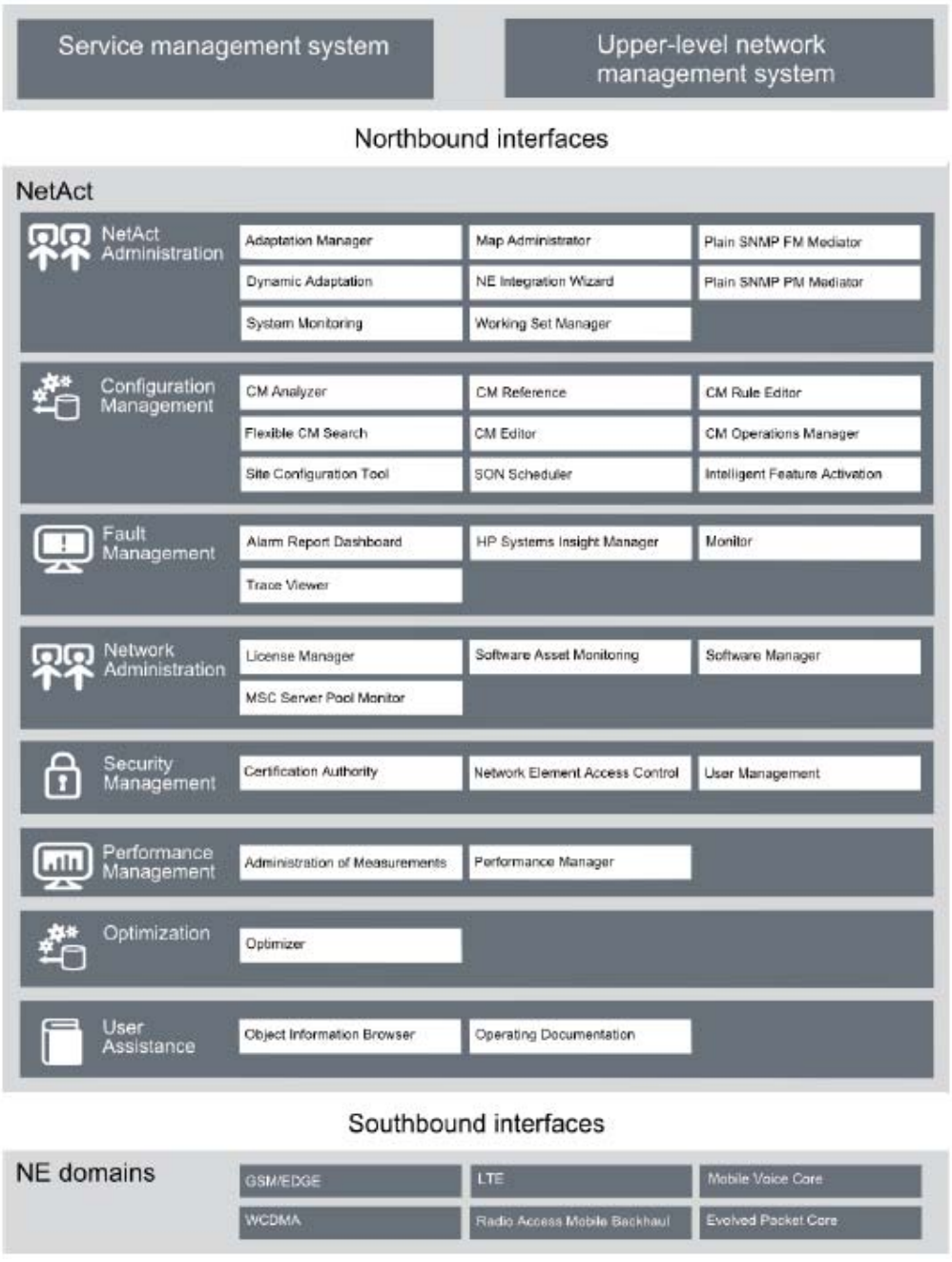
4.2 Toiminnot ja sovellukset

NetActin toiminnot on jaettu kuvan 3 mukaan seuraaviin ryhmiin:

- NetActin hallinta (NetAct administration)
- Konfiguraatioiden hallinta (Configuration management)
- Vian hallinta (Fault management)
- Verkon hallinta (Network administration)
- Turvallisuuden hallinta (Security management)
- Suorituskyvyn hallinta (Performance management)

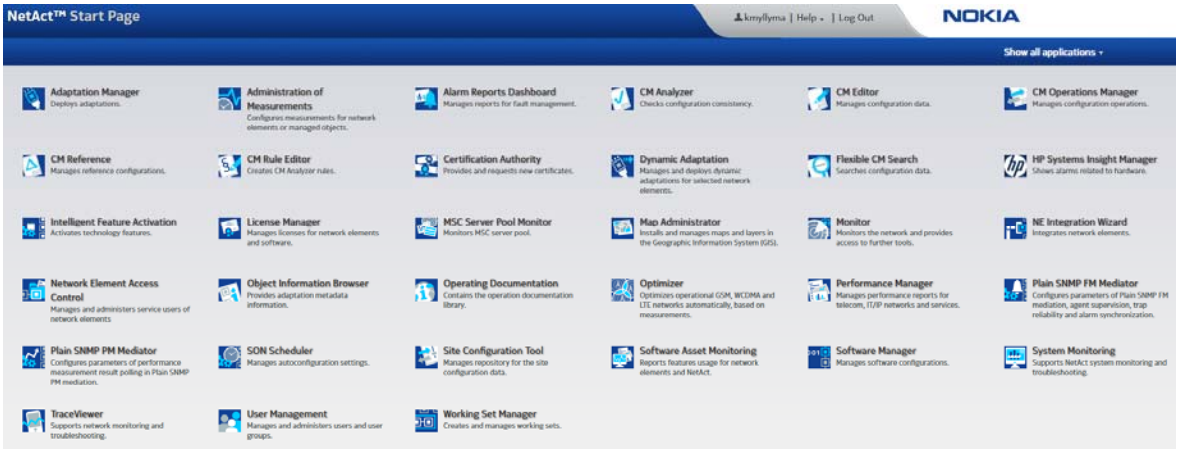
- Optimointi (Optimization)
- Käyttäjän avustaminen (User Assistance)

(Nokia NetAct 16.2 System Overview 2016, 4.)



KUVA 3. NetActin toiminnot ja sovellukset (Nokia NetAct 16.2 System Overview 2016, 5)

Kunkin toiminnon sisältä löytyy erilaisia sovelluksia, joiden käynnistysikonit löytyvät NetActin pääsivulta kuvan 4 tapaan:



KUVA 4. NetAct 16.2 pääsivun näkymä ja sovellusten käynnistysikonit

Osa NetActin pääsivulla näkyvistä sovelluksista on toteutettu HTML-kielellä, jolloin käynnistysikonin klikkauksen jälkeen sovellus avautuu verkkoselaimen uuteen ikkunaan ja sovellusta käytetään sen kautta. Osa sovelluksista on taas toteutettu Javalla, jolloin käynnistysikonin klikkauksen jälkeen käyttäjän tietokoneelle latautuu ensin sovelluksen sisältävä jnlp-tiedosto, joka sitten käynnistyy erillisenä sovelluksena.

NetActin käyttäjän työasemalle asetetaan tiettyjä teknisiä vaatimuksia liittyen muun muassa tuettuihin käyttöjärjestelmiin ja verkkoselaimiin. Java-sovellusten käyttöä varten tulee työasemassa olla asennettuna JRE (Java Runtime Environment). (Nokia NetAct 16.2 System Overview 2016, 28.)

Tämän opinnäytetyön kannalta keskeiset NetActin toiminnot, niiden sovellukset sekä toteutustavat on esitetty taulukossa 1.

TAULUKKO 1. Eräitä keskeisiä NetActin toimintoja ja sovelluksia

Toiminto	Sovellus	Käyttötarkoitus	Toteutus
Konfiguraatioiden hallinta (Configuration management)	CM Editor	Parametrien tarkistus ja muokkaus	Java
	CM Operations Manager	Toimintojen valmistelu ja lataus verkkoon	Java
	Flexible CM Search	Verkkokonfiguraatioiden hallintatietojen hakeminen	HTML
Vian hallinta (Fault Management)	Monitor	Verkon monitorointi ja pääsy muihin työkaluihin	Java
Verkon hallinta (Network administration)	Software Manager	Ohjelmistokonfiguraatioiden hallinta	HTML

Tiettyjä NetActin toimintoja voi ajaa myös komentorivin kautta käyttäen niin sanottua komentorivi- eli CLI- (Command Line Interface) rajapintaa. Tämä tarjoaakin kolmannen teknisen toteutustavan joillekin testitapauksille myös testausautomaation näkökulmasta katsottuna.

5 NOKIA FLEXI MULTIRADIO 10

5.1 Nokia Flexi Multiradio 10:n kuvaus

Radioverkkojen tilaajamäärien ja liikenteen yhä kasvaessa, uusien tukiasemapaikkojen löytämisen vaikeutuessa ja verkkojen monimutkaistuessa markkinoilla on ollut tarve mahdollisimman modulaariselle, helposti asennettavalle ja ohjelmallisesti myös etäyhteyden kautta muunneltavalle tukiasemalle laitemoduuleineen ja ohjelmistoineen. Nokian ratkaisu tähän tarpeeseen on Flexi Multiradio 10 -tukiasema. Toisin kuin perinteiset kaappimaiset tukiasemat sisään asennettavine pistoyksiköineen, Nokia Flexi Multiradio 10 koostuu erillisistä fyysisistä moduuleista, joista operaattori voi rakentaa erilaisia tukiasemia ja -konfiguraatiota. Moduuleita voi asentaa päällekkäin tai erilaisten asennussarjojen avulla vaikkapa putkeen, mastoon ja seinälle.

Nokia Flexi Multiradio 10 -tuoteperheen laitemoduulit voi jakaa toiminnallisuuden perusteella kahteen pääryhmään eli digitaaliseen signaalinkäsittelyyn keskittyvät ja radioverkkoon siirtoverkon kautta kytkettävät systeemimoduulit sekä radiotaajuisen ilmarajapinnan käsittelyyn tarvittavat radiomoduulit.

Kuvassa 5 on tyypillinen Flexi Multiradio 10 -tukiasema, jonka laitemoduulit on asennettu päällekkäin pinoon.

KUVA 5. Tyypillinen Nokia Flexi Multiradio 10 -tukiasema

5.2 Ohjelmisto

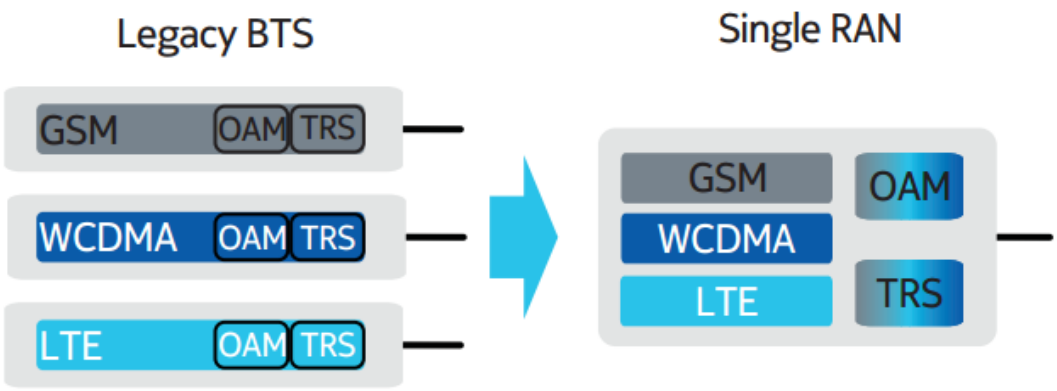
Flexi Multiradio 10 -tuoteperheen systeemimoduuli ja siihen kytketyt radiomoduulit voivat toimia perinteisenä yhden radioteknologian GSM-, WCDMA- tai LTE-tukiasemana, jos käytetään vain kyseistä radiotekniikkaa tukevaa ohjelmistoa. Tällaisesta yhden radiotekniikan tukiasemajärjestelmästä käytetään usein lyhennettä SRAT (Single Radio Access Technology).

Ohjelmistovaihdon ja uudelleen konfiguroinnin myötä samat Flexi Multiradio 10 -tuoteperheen systeemimoduuli ja siihen kytketyt radiomoduulit muodostavat SRAN/SBTS- (Single RAN BTS) tukiaseman, joka voi konfiguroinnista riippuen tukea yhtä, kahta tai jopa kaikkia kolmea radioteknologiaa yhtä aikaa.

6 SRAN-VERKKO JA SBTS-TUKIASEMA

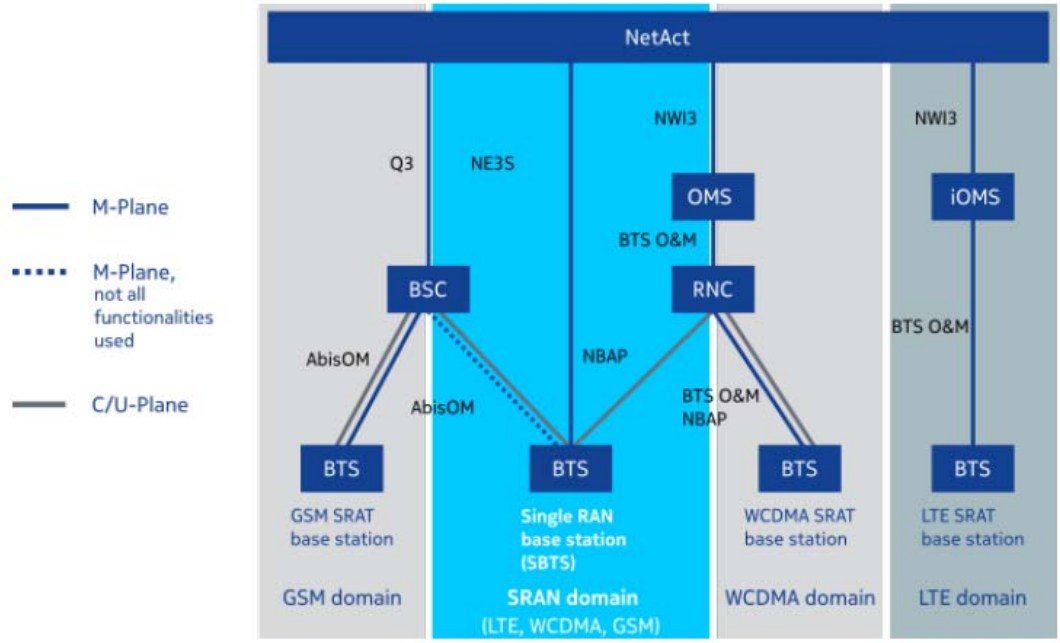
Keskeisiä osia minkä tahansa tukiaseman ohjelmistossa ovat OAM- (Operation and Maintenance) ja TRS- (Transmission) lohkot, jotka huolehtivat tukiaseman tärkeimmistä toiminnoista kuten käyttöliittymästä, ohjelmiston ja konfiguraatioiden hallinnasta, tukiaseman valvonnasta, hälytyksistä sekä liitynnästä tukiasemaohjaimiin ja runkoverkkoon. Perinteisen (legacy) SRAT-tukiaseman OAM- ja TRS-ohjelmistolohkot ovat suunniteltuja toimimaan vain yhdellä radiotekniikalla. SBTS-tukiaseman ohjelmistossa OAM- ja TRS-lohkot ovat yhteisiä kaikille samassa tukiasemassa toimiville eri radioteknologioille. Operaattorin kannalta katsottuna tämä mahdollistaa helpomman ja yhtenäisen verkon hallinnan verrattuna perinteiseen radioverkkoon.

Ero näkyy selkeästi kuvassa 6, jossa vasemmalla puolella on kolme perinteistä SRAT-tukiasemaa GSM-, WCDMA- ja LTE-radioteknologioille ja oikealla vastaavasti yksi SRAN/SBTS-tukiasema, joka tukee kaikkia kolmea radioteknologiaa yhtä aikaa.



KUVA 6. SRAT- ja SRAN/SBTS-tukiasemat (Nokia Single RAN System Description 16.2 2016, viitattu 28.11.2016)

Ero näkyy hyvin myös koko radioverkon arkkitehtuurin ja NetActin kannalta katsottuna. Kuvassa 7 on tyypillinen radioverkko, jossa NetActiin on integroitu sekä perinteisiä SRAT- että uusia SRAN/SBTS-tukiasemia.



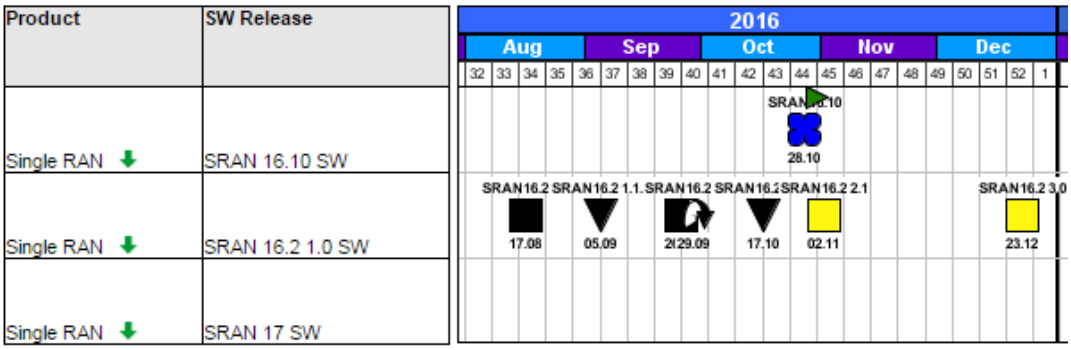
KUVA 7. Radioverkko, jossa SRAT- ja SRAN/SBTS-tukiasemia (Nokia NetAct 16.2 Nokia Single RAN Management Overview 2016, 5)

6.1 SRAN/SBTS-ohjelmistojulkaisut

Nokian luontiprosessi (Create Process) määrittelee ohjelmistojulkaisujen jakelutyypin ja -sisällön sekä julkaisujen nimeämiskäytännön. Nokia SRAN/SBTS -ohjelmistot on jaettu eri julkaisuihin (releases) niiden sisältämien ominaisuuksien (features) perusteella. Ohjelmistojulkaisujen nimeäminen pohjautuu niiden julkaisuvuoteen ja -kuukauteen, jolloin esimerkiksi SRAN16.2 viittaa helmikuuhun vuonna 2016 ja vastaavasti SRAN16.10 saman vuoden lokakuuhun. Tietyn ohjelmiston ensijulkaisun jälkeen siitä julkaistaan myöhemmin uusia versiota, jotka sisältävät korjauksia löytyneisiin vikoihin ja joihin lisätään puuttuvia toiminnallisuuksia. Riippuen uuden julkaisun sisällöstä ja sen merkityksestä asiakkaalle, julkaisuista käytetään erilaisia nimityksiä kuten esimerkiksi Standard Update, Priority Package ja Security Update. (NSN Create Process – Delivery Types and Naming Policy 2016, 5.)

Asiakkaat saavat ladattua julkaistut ja heidän kanssaan tehtyjen sopimusten mukaiset ohjelmistot Nokia Networks Online Services -sivuston kautta. Siellä voi nähdä myös jo julkaistujen ja edelleen tuettujen ohjelmistojen lisäksi tulossa olevien ohjelmistojen julkaisupäivämäärät. (Nokia Networks Online Services 2016, viitattu 28.11.2016.)

Kuvassa 8 näkyy tämän työn kirjoitushetkellä tiedossa olevat SRAN/SBTS ohjelmistojulkaisut vuonna 2016.



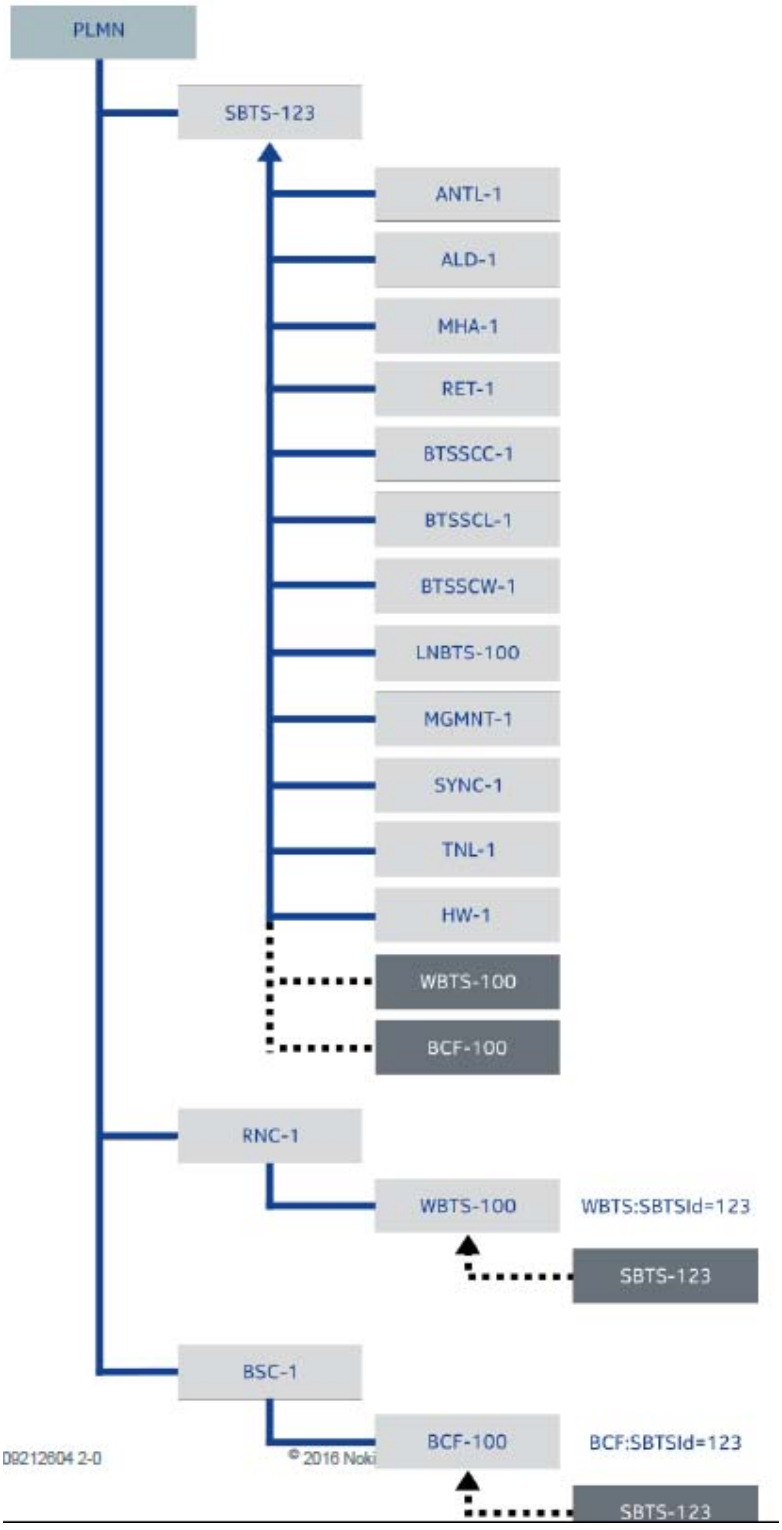
KUVA 8. SRAN/SBTS ohjelmistojulkaisut vuonna 2016

Tämän opinnäytetyön yhteydessä kehitetty testausautomaatio keskittyy SRAN16.2-julkaisuun, jonka ensimmäinen asiakasversio oli saatavilla helmikuussa 2016.

6.2 SRAN/SBTS-verkon arkkitehtuuri

SRAN OAM -verkkoarkkitehtuuria kuvataan hierarkisella oliomallilla, jossa ylimpänä oliona on PLMN (Public Land Mobile Network) kuvaten koko verkkoa. SBTS-tukiasemaa kuvaa oma olio ja jokaisella oliolla on yksilöllinen SBTS ID -tunniste. SBTS sijaitsee PLMN:n alapuolella ja SRAT LTE -tukiasemaa kuvaava LNBTS taas suoraan SBTS-olion alla.

WCDMA- ja GSM-tukiasemia kuvaavat oliot, WBTS ja BCF, sijaitsevat vastaavasti niitä ohjaavien tukiasemaohjaimien, RNC ja BSC, alla. RNC/WBTS- ja BSC/BCF-radiokonfiguraatioiden suhde SBTS-olioon määritellään niin sanotuilla sukulaisuusattribuuteilla. SBTS-tukiasema kertoo käyttöönoton yhteydessä oman SBTS ID:n sekä RNC:lle että BSC:lle ja näiden avulla NetAct voi luoda suhteen SBTS-olion ja tukiasemaohjaimien alla olevien WCDMA- ja GSM-radioverkkokonfiguraatioiden välillä. Kuva 9 esittää SRAN/SBTS-oliomallia, jossa yhtenäinen viiva esittää suoraa suhdetta ja katkoviiva edellä mainittua sukulaisuussuhdetta SBTS-olioon. (Nokia NetAct 16.2 Single RAN Management Overview 2016, 7-8.)



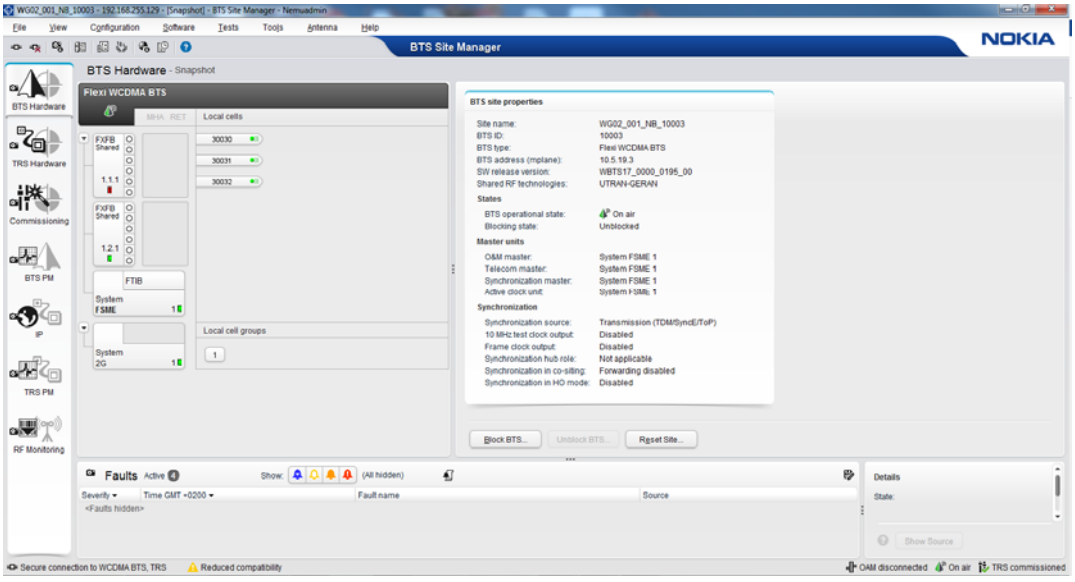
KUVA 9. SRAN/SBTS oliomalli (Nokia NetAct 16.2 Single RAN Management Overview 2016, 8)

Olioista käytetään nimitystä Managed Object (MO) ja jokaiselle niistä on yksiselitteinen tunniste, Distinguished Name (DN). DN määrittää olion suhteen sen vanhempiin sekä olion luokan. Esimerkkinä vaikkapa DN-tunniste PLMN-PLMN/SBTS-5268/LNBTS-5268/LNCEL-18609, jossa alimmalla tasolla olevan LTE-solun LNCEL-18609 suhde näkyy sen yläpuolisiin vanhempiin.

6.3 SBTS-tukiaseman käyttöliittymä

Kuten aiemmin jo todettiin, tukiasema on eräs mobiiliverkon elementti. Tukiaseman konfigurointiin ja hallintaan käytettävästä sovelluksesta käytetäänkin tämän vuoksi yleisesti BTS Element Manager -nimeä. Perinteisissä SRAT-tukiasemissa BTS Element Manager on erillinen käyttäjän tietokoneelle asennettava sovellus. Paikallinen BTS Element Manager -yhteys SRAT-tukiasemaan luodaan kytkemällä käyttäjän tietokoneesta Ethernet-kaapeli suoraan tukiaseman systeemimoduuliin. Tukiasemaan voi ottaa myös etäyhteyden, jolloin BTS Element Manager -yhteys luodaan käyttämällä tukiaseman yksilöllistä IP-osoitetta. Etäyhteyden aikana käytetään tukiaseman ja radioverkon välistä transmissioyhteyttä.

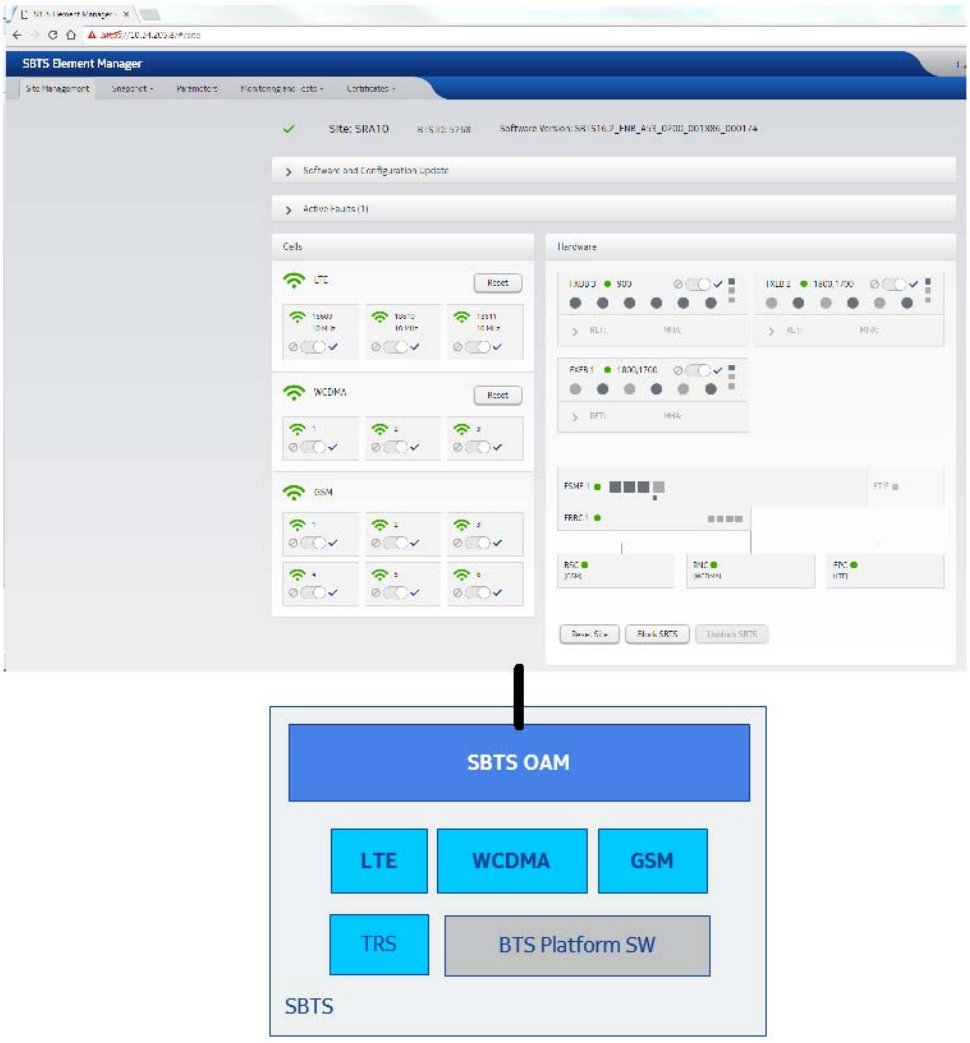
Kuvassa 10 on tyypillinen perinteisen SRAT-tukiaseman BTS Element Manager -näkymä graafisine symboleineen.



KUVA 10. Tyypillinen SRAT-tukiaseman BTS Element Manager -näkymä

SBTS-tukiaseman kanssa erillistä BTS Element Manager -sovellusta ei tarvita vaan tukiasemaa hallitaan sen sisäänrakennetun käyttöliittymän kautta. SBTS:n käyttöliittymä tunnetaan yleisesti nimellä WebUI, joka on toteutettu SBTS-tukiaseman sisäisessä OAM-ohjelmistolohkossa. WebUI:n käyttöön riittää normaali verkkoselain. Kuten SRAT-tukiaseman tapauksessakin, myös SBTS-tukiaseman WebUI-yhteyden voi luoda joko paikallisesti tai etäyhteyden kautta. (Nokia Single RAN 16.2. Migration to SBTS and Commissioning 2016, 10.)

WebUI:n tyypillinen näkymä sekä SBTS-ohjelmiston lohkokaavio on esitetty kuvassa 11.



KUVA 11. Tyypillinen WebUI-näkymä sekä SBTS-ohjelmiston lohkokaavio

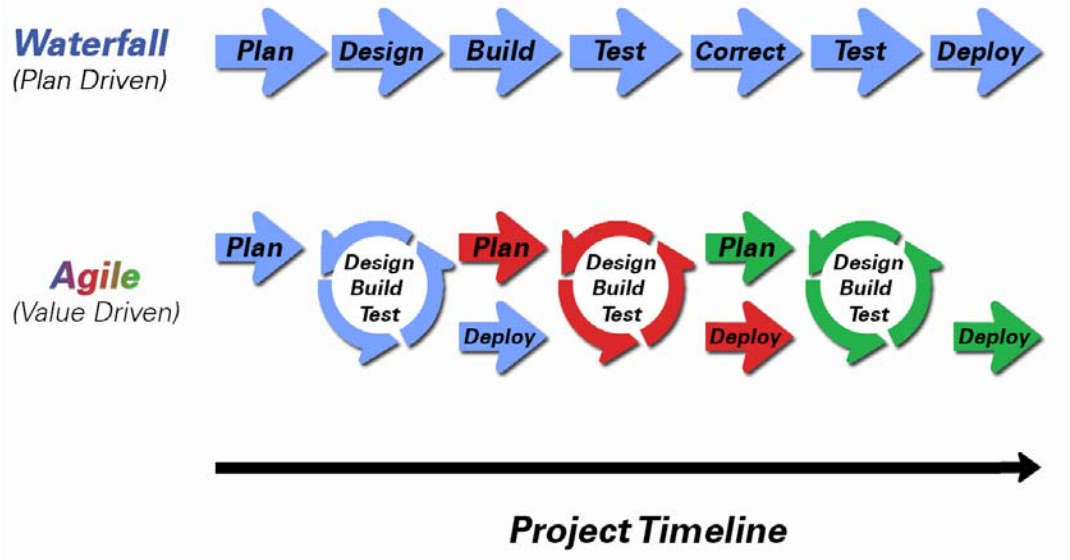
7 OHJELMISTOJEN KEHITYS JA TESTAUS

7.1 Yleistä ohjelmistojen kehityksestä

Modernit ohjelmistojen kehitystavat voidaan jakaa perinteiseen (traditional) ja ketterään (agile) menetelmään. Perinteisessä menetelmässä ohjelmiston kehitysvaiheet ovat selkeitä kokonaisuuksia ja seuraavat ajallisesti toisiaan. Alussa ohjelmistolle tehdään vaatimusmäärittely, sen jälkeen suunnittelu ja koodaus, koodin testaus ja lopulta käyttöönotto. Menetelmän ytimessä on yksityiskohtainen näkemys lopputuotteesta ennen sen käyttöönottoa. Ketterässä (agile) menetelmässä ytimessä on vähittäinen ja iteratiivinen kehitys, jossa prosessin eri vaiheita tehdään uudelleen. Kehittäjien näkökulmasta ohjelmisto ei ole laaja vaan orgaaninen kokonaisuus monine muuttuvine osineen, jotka vaikuttavat toisiinsa. Tämän vuoksi joustavuus ja jatkuva testaaminen ovat tärkeitä. (Optimus Information 2016, viitattu 28.11.2016.)

Perinteinen menetelmä on prosessina hidas eikä sen vuoksi välttämättä sovellu käytettäväksi uusissa kaupallisissa ohjelmistoprojekteissa, joissa markkinoille pääsyn nopeus voi olla ratkaisevan tärkeää tuotteen kaupallisen menestyksen kannalta. Nokian tuoteprosessissa mainitaan molemmat menetelmät, joista suositellaan käytettäväksi ketterää menetelmää silloin, kun vaatimukset ja prioriteetit muuttuvat tai kun kehitetään uudentyyppistä tuotetta. (Nokia Create Process – SW Development 2016, 3.)

Peräkkäisistä vaiheista johtuen perinteistä menetelmää kutsutaan usein vesiputousmalliksi (Waterfall). Kuvan 12 yläosassa kuvataan perinteistä menetelmää peräkkäisine vaiheineen ja kuvan alaosassa vastaavasti ketterää mallia lyhyine iteraatioineen.



KUVA 12. Perinteinen sekä ketterä ohjelmistonkehitysmalli (Vogel 2014, viitattu 28.11.2016)

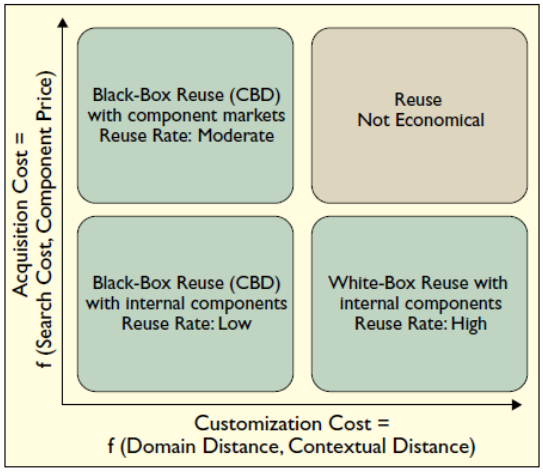
Kuvassa 12 on helppo havaita tiettyjä ketterään kehitystapaan liittyviä olennaisia seikkoja, kuten ohjelmiston jatkuva kehittäminen, sen jatkuva testaus eri kehitysvaiheissa sekä testausten mahdollisimman pitkälle viety automatisointi. Molempiin menetelmiin kuuluu olennaisena osana myös löytyneiden vikojen korjaus (Correct), vaikka se on käytetyn lähteen kuvassa merkitty vain perinteiseen menetelmään.

7.2 Ohjelmiston uudelleenkäyttö

Ohjelmiston uudelleenkäytöllä tarkoitetaan aikaisemmin kehitetyn ohjelmiston käyttöä useammassa projektissa (Ravichandran & Rothenberger 2003, 109). Uudelleenkäytettävän ohjelmiston voidaan määritellä tarkoittavan ohjelmistoa, joka on suunniteltu ja toteutettu jotain tiettyä tarkoitusta ja uudelleenkäyttöä varten (Katz, Dabrowski, Miles & Law 1994, 11). Määritelmän mukaan uudelleenkäytettävyys on siis pitänyt huomioida jo ohjelmiston suunnitteluvaiheessa. Uudelleenkäytettävyydellä taas tarkoitetaan sitä, kuinka hyvin ohjelmistoa voidaan käyttää uudelleen useammassa kuin yhdessä ohjelmistojärjestelmässä. (Katz ym. 1994, 11). Ohjelmistoa voidaan käyttää uudelleen eri tasoilla. Tässä yhteydessä hyvin soveltuva jako on ohjelmiston uudelleenkäyttö joko sellaisenaan, komponenttitasolla tai yksittäisen funktion tai objektin tasolla (Sommerville 2004, 5).

Myös ohjelmiston uudelleenkäytön strategiaa voidaan tarkastella aiemmin ohjelmiston testauksen yhteydessä esiteltyjen white- ja black-box -konseptien avulla. Niiden etuja ja haittoja voidaan arvioida tarkastelemalla niitä kustannuksia, jotka kuluvat sekä hankintaan että tarvittavaan räätälöintiin. White-box -tapauksessa ohjelmiston lähdekoodi on käytettävissä, jolloin hankintakulut jäävät pieniksi ja komponenttia muokkaamalla siitä voidaan räätälöidä uuteen käyttötarkoitukseen sopiva. Tarvittavien muokkausten tekeminen edellyttää kuitenkin lähdekoodin yksityiskohtien syvällistä ymmärtämistä, mistä seuraa muutoskulujen kasvaminen. Black-box -tapauksessa käytetään markkinoilta valmiina löytyviä komponentteja. Ohjelmistokomponentin lähdekoodi ei ole käytettävissä eikä muutoksista aiheutuvia kuluja voi näin syntyä. Jos sopivaa komponenttia joudutaan etsimään markkinoilta, nousevat hankintakulut suuremmaksi mutta toisaalta löytyvän komponentin uudelleenkäytettävyys voi olla parempi. White-box -koodin muokkausta ulkoisesti hankittuna ei nähdä taloudellisesti järkeväksi. (Ravichandran & Rothenberger 2003, viitattu 28.11.2016.)

Kuva 13 esittää edellä kuvattua ohjelmiston uudelleenkäytön strategiaa, jossa pystyakseli kuvaa tarvittavien ohjelmistokomponenttien hankkimiskuluja ja vaaka-akseli komponentteihin mahdollisesti tarvittavien muutosten aiheuttamia kustannuksia.



KUVA 13. Ohjelmiston uudelleenkäytön strategia (Ravichandran & Rothenberger 2003, viitattu 28.11.2016)

Spolskyn (2000, 1) mukaan ohjelmiston tekeminen puhtaalta pöydältä on suurin yksittäinen strateginen virhe, minkä ohjelmistoyritys voi tehdä. Esimerkkinä tästä hän mainitsee Netscape 6.0 -verkkoselaimen kehityksen, joka koodattiin tyhjästä ja edellinen versio ehti olla markkinoilla lähes kolme vuotta. Viiveen vuoksi Netscape menetti markkinaosuutensa.

7.3 Yleistä ohjelmistojen testauksesta

Ohjelmistojen testaukseen viitataan ja sitä kuvataan useissa lähteissä ja muun muassa Endresin ja Rombachin (2003, 123) mukaan se on olennainen osa ohjelmistotuotteen kehitysprosessia ja toisaalta avainmenetelmä ohjelmiston verifiointiin ja virheiden löytämiseen. Toisaalta jos ohjelmistokehitys olisi täydellistä, ei virheitäkään teoriassa voisi syntyä ja näin testaus olisi turha ja resursseja tuhlaava toimenpide.

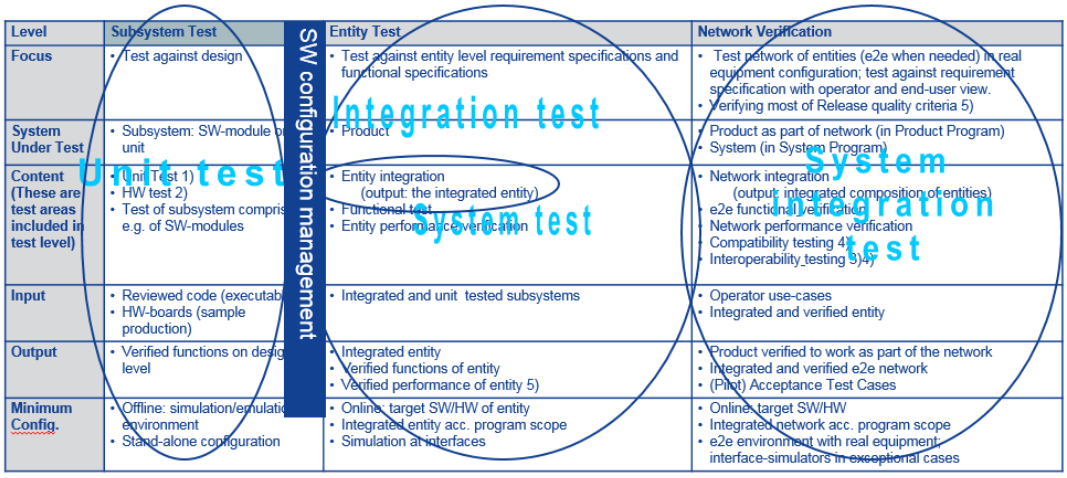
On kuitenkin helppo yhtyä Endresin ja Rombachin (2003, 125) näkemykseen siitä, että testausta ei voida koskaan jättää tekemättä riippumatta siitä, miten huolellisesti ohjelmiston suunnittelu, kehitys tai analysointi on tehty. Testauksen tärkeydestä ja tarvittavasta kattavuudesta on tosin olemassa monia mielipiteitä ja näkemyksiä, sillä testaus vääjäämättä vaatii resursseja ollen näin verraten suuri kulueräkin. Ohjelmistoissa on kuitenkin aina virheitä, joista mahdollisimman moni tulee löytää ja korjata ennen lopullisen ohjelmistoversion julkaisua. Brittonin, Jengin, Carverin, Chearkin ja Katzenellenbogenin (2012, 6) mukaan ohjelmistojen kehittäjät käyttävätkin puolet ajastaan virheiden korjaamiseen.

Nokian sisäisessä materiaalissa todetaan osuvasti, että testauksella voidaan löytää vikoja, mutta testaamalla ei voi todistaa, etteikö niitä olisi. Täysin kattava testaus ei käytännössä ole mahdollista eli haasteena onkin löytää ne testitapaukset, jotka olisivat mahdollisimman kattavia. Testauksen tavoitteet on määritelty seuraavasti: ”Target is to find the most important defects first and to do the best possible testing in the time that is available”. Samaisen lähteen mukaan testauksen suunnittelun pitäisi perustua riskitason arviointiin, jossa riskiä arvioidaan löytyneen vian aiheuttaman vaurion suuruuden ja toisaalta vian todennäköisyyden perusteella. (Nokia Create Process – General testing overview 2013, 7.)

Ohjelmiston testaus voidaan jaotella monella eri tavalla. Loveland, Miller, Prewitt ja Shannon (2005, 28–43) jakavat testauksen eri vaiheisiin, joita ovat yksikkötestaus (Unit Test), toiminnallinen testaus

(Function Verification Test), järjestelmätason testaus (System Verification Test), suorituskyvyn testaus (Performance Verification Test), integrointitestaus (Integration Test) ja beetatestaus (Beta Test). Nokian sisäiset testivaiheet on jaettu osajärjestelmä- (Subsystem Test), kokonaisuus- (Entity Test) ja verkko- (Network Verification) tasoihin kuvan 14 mukaan. Kuvassa on soikioilla korostettu eri tasoihin kuuluvia testausvaiheita. Kuvan mukaan osajärjestelmätasolle kuuluu muun muassa yksikkötestaus (Unit Test), kokonaisuustasolle integrointitestaus (Integration Test) ja järjestelmätestaus (System Test) sekä verkkotasolle järjestelmän integrointitestaus (System Integration Test). (Nokia Create Process – General testing overview 2013, 10.) Nimeämistapoja on siis erilaisia ja tärkeintä onkin ymmärtää, mitä kukin vaihe pitää sisällään kussakin tapauksessa.

Tässä yhteydessä käytetään myös termejä SyVe- (System Verification) ja e2e- (end-to-end) testaus, jotka kuuluvat verkkotason testausvaiheeseen kuvan 14 mukaan. Testaus suoritetaan silloin todellisessa verkkoympäristössä ja testauksessa keskitytään verkko-operaattorin kannalta tärkeisiin testitapauksiin.



KUVA 14. Testauskategoriat ja -vaiheet (Nokia Create Process – General testing overview 2013, 10)

Testaus voidaan jakaa myös sen mukaan, onko ohjelmiston lähdekoodi tunnettu vai ei. Niin sanotussa black-box -tilanteessa testaus perustuu vain määrityksiin eikä mitään muuta oleteta. Black-box -testaus tunnetaan myös vaatimuspohjaisena (specification-based) tai toiminnallisena testauksena (functional testing). Oleellista tässä on siis se, että testaaja tietää miten ohjelmiston pitäisi toimia mutta ei tiedä, mitä koodissa tapahtuu tai miten koodi on toteutettu. White-box -testauksessa ohjelmiston lähdekoodi tunnetaan ja testaaja voi nähdä koodin toteutuksen ja valita testitapaukset

sen mukaan. White-box -testaus tunnetaan myös nimillä koodipohjainen (program based) tai rakenteinen testaus (structured testing). (Endres & Rombach 2003, 124–125.) Nokian sisäisessä koulutusmateriaalissa mainitaan myös grey-box -tapaus, joka on jossain edellä mainittujen vaihtoehtojen välimaastossa (Nokia Create Process – General testing overview 2013, 20).

Tässä yhteydessä keskitytään järjestelmätason testaukseen, joka perustuu vaatimuksiin. Eri verkkoelementtien ohjelmistokoodia tai niiden toteutusta ei tunneta eli sen perusteella kyse on black-box -testauksesta. Testaaja voi kuitenkin tutkia verkkoelementtien ja ohjelmistojen sisäisten rajapintojen toimintaa ja niiden välistä liikennettä tuotekehitystyökaluilla, joten tämän perusteella kyseessä voisi olla myös edellä mainittu grey-box -testaus.

7.4 SRAN/SBTS-testausstrategia

SRAN/SBTS-järjestelmätason testausstrategiassa määritellyistä asioista keskeisimpiä tässä yhteydessä ovat seuraavat:

- Testaustiimin tulee määritellä automaattisesti testattavat alueet ja kaikkein tärkeimmät testitapaukset tulee automatisoida.
- Automatisoiduista testitapauksista tulee luoda CRT1- ja/tai CRT2-testikokoelmat.
- Työn tehostamiseksi automaation kehitykseen ja ajoon tulisi käyttää nimettyjä henkilöitä.
- Testaustulokset pitäisi olla saatavilla keskitetysti tietokannasta.
- Testausautomaatiossa tulisi mahdollisuuksien mukaan käyttää olemassa olevia automaatiotyökaluja ja – testitapauksia.

(Nokia SRAN 16.2 Master Test Plan 2015, 40.)

Mobiiliverkoissa välitettävän liikenteen muutoksista ja verkko-operaattoreiden tarpeista johtuen erilaisia SRAN/SBTS:ssä tuettuja konfiguraatioita ja ohjelmiston ominaisuuksia on paljon ja niiden määrä on kasvussa. Ennen uuden konfiguraation tai ohjelmiston ominaisuuden julkistamista ja virallista tukea ne tulee testata, mikä rajalliset resurssit huomioiden aiheuttaa kasvavaa tarvetta suorittaa testausta automaattisesti.

Kuten edellä kuvattiin, olennainen osa ketterää ohjelmistokehitystä on jatkuva testaus ohjelmistokehityksen eri vaiheissa. Eräs testauksen laji on regressiotestaus, jolla pyritään varmistamaan oh-

jelmistoihin aiemmin toteutettujen toiminnallisuuksien virheetön toiminta myös uusissa ohjelmistoversioissa (Dustin, Rashka & Paul 1999, 45). Tässä yhteydessä tärkeimpänä sovelluskohteena testausautomaatiolle onkin järjestelmätason regressiotestaus.

Ketterän ohjelmistokehityksen yhteydessä käytetään usein lyhennettä CRT (Continuous Regression Testing), jolla viitataan jatkuvaan regressiotestaukseen. Tässä yhteydessä keskeisiä termejä ovat myös CRT1 ja CRT2, joilla tarkoitetaan kahta erilaista testikokoelmaa. Sekä CRT1- että CRT2-testikokoelmat sisältävät erilaisia testitapauksia. CRT1-testikokoelman sisältämien testitapausten kokonaisajoajan tulisi olla lyhyempi kuin CRT2-testikokoelman. CRT2-testikokoelman ajo aloitetaankin tyypillisesti vasta sen jälkeen, kun CRT1-ajo on mennyt läpi. CRT1- ja CRT2-testikokoelmiin kuuluvia testitapauksia ei ole tarkasti määritelty vaan ne voivat vaihdella tarpeen mukaan. Jokainen testaustiimi vastaa siitä, että CRT1- ja CRT2-kokoelmissa on mukana kaikkein tärkeimmät testitapaukset. (Nokia SRAN 16.2 Master Test Plan 2015, 34–37.)

Tyypillisiä CRT1- ja CRT2-testikokoelmiin kuuluvia testitapauksia ovat edellä mainitut erilaiset tukiaseman ja sen eri olioiden uudelleenkäynnistykset (reset) sekä tukiaseman ohjelmiston päivitykset. Jokaiseen testikokoelmaan kuuluu olennaisena osana sen varmistaminen, että tukiasema toipuu takaisin toimintaan ja tukiaseman sisältämien olioiden toiminnalliset tilat palaavat takaisin aktiivisiksi (enabled). Juuri toiminnallisten tilojen tarkistukseen liittyvät testitapaukset ovat tämän opinnäytetyön keskiössä.

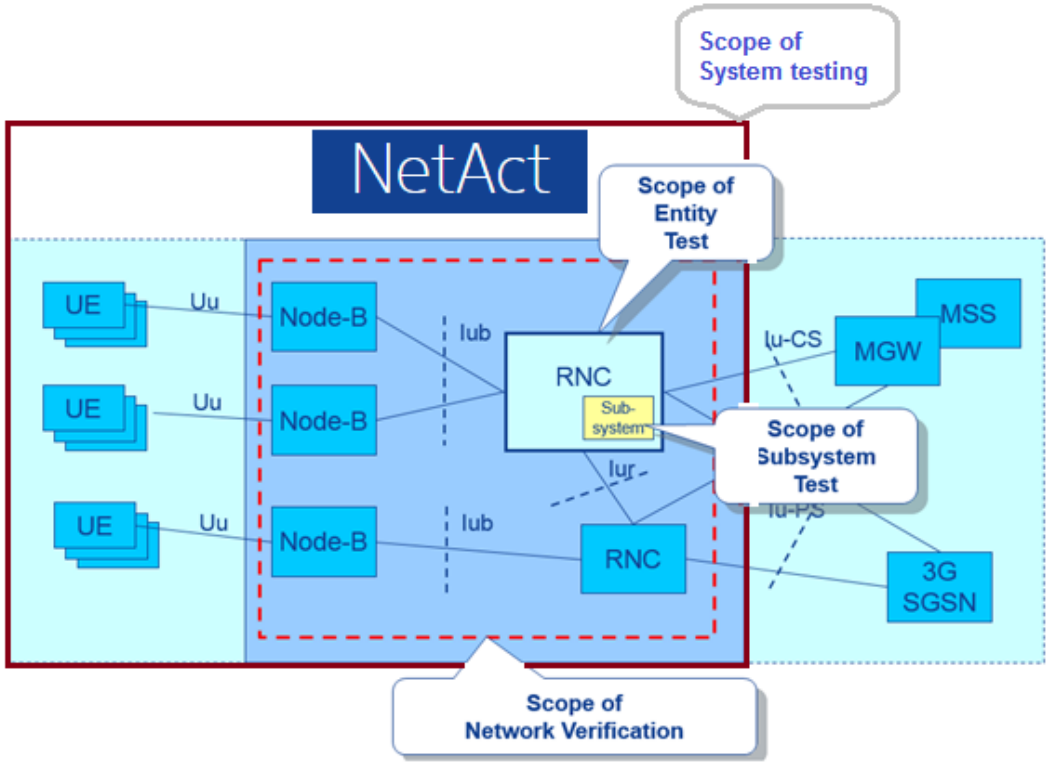
7.5 SRAN/SBTS-järjestelmätason testaus

Nokian sisäisessä Create-prosessissa järjestelmätason testaus määritellään alueeksi, jossa testit suoritetaan oikeassa verkkoympäristössä verkko-operaattorin ja loppukäyttäjän kannalta katsottuna. Järjestelmätason testauksessa tulee testata erityisesti verkko-operaattorin tyypillisiä verkkojen toiminnallisuuteen ja hallintaan liittyviä operaatiota. (Nokia Create Process – Integration and Verification (I&V): Training, test levels and areas part 2016, 5.)

Edellä esitettyjen testaustasojen eron havaitsee selvästi, kun tasoja tarkastellaan mobiiliverkon elementtejä vasten. Kuvassa 15 on osajärjestelmätestauksen (Subsystem Test) kohteena RNC:n sisäinen ohjelmistolohko, yksikkötestauksessa (Entity Test) koko RNC ja verkkotason (Network Verification) testauksessa 3G-radioverkko, johon kuuluvat RNC:n lisäksi myös 3G-tukiasemat eli

NodeB:t (Nokia Create Process – Integration and Verification (I&V): Training, test levels and areas part 2016, 4).

Tämän työn yhteydessä keskeisiä termejä ovat järjestelmätestaus (System Testing) ja järjestelmän verifiointi (System Verification), jotka edellisen kuvan 14 mukaan kuuluvat verkkotestauksen (Network Verification) kategoriaan. Verkkotestauksen laajuus voi vaihdella koko verkon kattavasta e2e-testauksesta minimissään kahden verkkoelementin väliseen testaukseen. Tähän työhön liittyvä järjestelmätestauksen laajuus (Scope of System Testing) on lisätty alkuperäiseen kuvaan selkeyden vuoksi. Radioverkon laitteiden lisäksi testattavaan järjestelmään kuuluvat silloin myös päätelaitteet (UE) sekä verkon hallinnointiin käytettävä NetAct.



KUVA 15. Esimerkki testaustasojen eroista. (Nokia Create Process – Integration and Verification (I&V): Training, test levels and areas part 2016, 4)

8 TESTAUSAUTOMAATIO

8.1 Yleistä testausautomaatiosta

Organisaatiot haluavat yhtäältä testata ohjelmistoja riittävästi mutta toisaalta mahdollisimman lyhyessä ajassa, jolloin katseet kääntyvät testausautomaation puoleen. Testausautomaatio voidaankin määritellä prosessina, jossa testausaktiviteettien hallinta sekä automaattisten komentosarjojen kehitys ja ajaminen tehdään automaattisella testaustyökalulla, tavoitteena testivaatimusten todentaminen. Testausautomaatio voi tuottaa parhaan hyödyn silloin, kun testauksessa käytettäviä komentosarjoja tai niiden osia voi tai pitää toistaa usein. Erityisen hyödyllinen testausautomaatio on järjestelmätason regressiotestauksessa. (Dustin, Rashka & Paul 1999, 4.)

Dustin, Garrett ja Gauf (2009, 4) määrittelevät testausautomaation laveammin ja heidän mukaansa se käsittää kaikki testausvaiheet. Heidän mukaansa kaikki ne testausvaiheet, jotka tehdään käsin, voidaan haluttaessa myös automatisoida. Heidän mielestään manuaalinen testaus poikkeaa automaattisesta testauksesta neljällä eri tavalla. Ensinnäkin automaattisella testauksella voidaan suorittaa tehtäviä, joita manuaalisesti ei välttämättä saataisi tehtyä. Toisaalta testausautomaation kehittäminen on myös osaltaan ohjelmistokehitystä. Kolmantena seikkana he mainitsevat, että automaattinen testaus ei vähennä tarvetta testistrategian luomiseen, tulosten analysointiin ja testaustekniikoiden ymmärtämiseen vaan niissä tarvitaan edelleenkin ihmistä ja manuaalista työtä. Viimeisenä seikkana he tuovat esiin sen, että molempia testaustapoja tarvitaan, sillä ne täydentävät toisiaan.

Piiroinen (2006, 10–11) viittaa Fewsterin ja Grahamin (1999, 500) kirjaan mainitessaan, että testausautomaatiossa on kysymys sekä testauksesta että automaatiosta. Sen mukaan testauksessa on oleellista löytää suuri määrä virheitä rajallisella määrällä testitapauksia ja testaajan tulee tietää, mitä voidaan ja pitää testata ja miten. Automaatio on puolestaan ohjelmointia eikä näin eroa oleellisesti ohjelmistokehityksestä. Kanerin (2000, 3) mielestä pitäisikin mieluummin puhua tietokoneavusteisesta testauksesta.

8.2 Tavoitteet ja vaatimukset

Testausautomaatiota voidaan tarkastella sekä sen tavoitteiden, sille asetettavien vaatimusten että sen etujen kannalta. Testausautomaation ylätason tavoitteiksi voidaan määritellä seuraavat kuusi seikkaa, joista kolme ensimmäistä liittyvät itse testien tuomaan lisäarvoon ja loput kuvaavat itse testien ominaisuuksia:

- Testien tulisi parantaa laatua.
- Testien tulisi auttaa ymmärtämään testattavaa järjestelmää.
- Testien tulisi pienentää riskiä.
- Testejä tulisi olla helppo ajaa.
- Testejä tulisi olla helppo kirjoittaa ja ylläpitää.
- Testien ylläpitotarve on pieni, vaikka testattava järjestelmä kehittyy.

(Meszaros 2016. Goals of Test Automation, viitattu 28.11.2016.)

Testien tuomaan lisäarvoon liittyvistä seikoista erityisesti laadun paranemista tai riskin pienenemistä on syytä käsitellä tarkemmin, sillä ne eivät ole yksiselitteisiä asioita.

Meszarosen (2016. Goals of Test Automation, viitattu 28.11.2016) mukaan testien suunnittelu pakottaa miettimään eri skenaarioita riittävän syvällisesti ja tarkasti, mikä osaltaan parantaa myös ohjelmistomäärittelyn ja -vaatimusten laatua. Toisaalta ketterässä ohjelmistokehityksessä ajettavilla automaattisilla regressiotesteillä voidaan ajoissa havaita mahdollisia kertaalleen jo korjattuja vikoja ja näin estää tällaisia vikoja sisältävän koodiin päätyminen seuraaviin testivaiheisiin. Testattavan ohjelmiston laatua voi parantaa siis sekä vaatimusten tarkentuminen että mahdollisimman varhain havaitut viat.

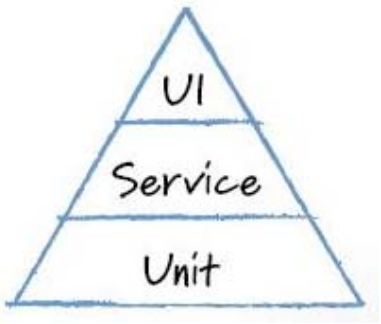
Ohjelmiston laadun paranemiseen liittyvät seikat vähentävät osaltaan myös ohjelmistovirheiden aiheuttamia riskejä. Toisaalta testausautomaatiolla suoritettava jatkuva regressiotestaus antaa mahdollisuuden kehittää ja tehdä muutoksia koodiin nopeammin eli testausautomaatiolla saatavat tulokset toimivat ikään kuin suojaverkkona. Hyvin osuvana vertauskuvana voi käyttää trapetsitaiteilijan alla olevaa suojaverkkoa, jota ilman temppujen harjoittelu ja niissä kehittyminen olisi käytännössä mahdotonta. Testausautomaatio voi tuoda myös uusia riskejä kuten vaikkapa sen, että testattavassa ohjelmistossa on sisäänrakennettua testauslogiikkaa vielä tuotantovaiheessakin. Si-

säänrakennetusta testauslogiikasta on usein hyötyä automaattisia testejä kehitettäessä mutta tuotantoympäristössä testauksen kohteena olevassa ohjelmistossa testauslogiikkaa ei saa olla sisäänkirjoitettuna. Testauslogiikka pitää rakentaa testausautomaation koodiin ja tarvittaessa sieltä voi ladata osia testattavaan kohteeseen testausajon aikana. (Meszarosen 2016. Goals of Test Automation, viitattu 28.11.2016.)

Testausautomaatiojärjestelmälle voidaan määritellä myös vaatimuksia, joiksi Laukkanen (2006, 106) luettelee testien ajon ilman valvontaa, järjestelmän helppokäyttöisyyden sekä ylläpidettävyyden. Nämä ovat hyvin samankaltaisia kuin aiemmin Meszarosen esittämällä listalla olleet kolme alinta seikkaa.

8.3 Testausautomaation tasot

Cohn (2009) esittelee artikkelissaan testausautomaatiolle kolme eri tasoa pyramidilla kuvan 16 mukaisesti.



KUVA 16. Testausautomaatiopyramidi (Cohn 2009)

Kuvan alimmalla tasolla on yksikkötestaus (Unit) ja Cohnin mukaan suurin osa testausautomaatiosta tulisi kehittää ja ajaa tällä tasolla, sillä ohjelmiston kehittäjä saa sieltä yksityiskohtaista tietoa mahdollisista koodivirheistä. Tämän tason voidaan tässä yhteydessä ymmärtää tarkoittavan aiemmin esiteltyä komentorivipohjaista eli CLI-testausta. Ylimmällä tasolla on vastaavasti käyttöliittymän UI- (User Interface) testaus, jolle testausautomaatiota tulisi kehittää ja käyttää mahdollisimman vähän, sillä vastaavat toiminnallisuudet on todennäköisesti testattu jo alemmilla tasoilla ja toisaalta testausautomaation kehittäminen tällä tasolla on kallista. Näiden ääripäiden väliin jää hänen mukaansa palvelukerros (Service), jonka käytännössä voidaan ymmärtää tarkoittavan erilaisten ohjelmistokokonaisuuksien testausta ilman, että niiden toiminnallisuutta testattaisiin lopullisen käyttöliittymän kautta. Samaan aiheeseen viittaa myös Francino (2015) artikkelissaan. Hänen mukaansa

käyttöliittymän automaattinen testaus on hidasta, testauskoodin ylläpito on vaikeaa ja koodi rikkoutuu helposti.

8.4 Testausautomaation edut ja haitat

Automaattisen testauksen hyödyt voidaan jakaa kolmeen luokkaan: Luotettavan järjestelmän tuottaminen, testauskattavuuden lisääntyminen sekä testauksen työmäärän pieneneminen ja ajan säästö. Näiden luokkien sisällä he luettelevat etuja tarkemmin. Tässä yhteydessä niistä oleellisimpia ovat automaattinen testaustulosten raportointi, tehokkaampi yhteensopivuus- ja regressiotestaus, pitkästyttävien ja arkipäiväisten testien ajo sekä ympärivuorokautinen testaus. Quality Assurance Instituten tekemän analyysin mukaan automaattinen testaus vaatii vain 25% siitä työmäärästä, jonka sama testaus vaatii manuaalisesti. (Dustin, Rashka & Paul 1999, 37–51.)

Ghahrain (2015, viitattu 28.11.2016) mukaan testausautomaation etuja ovat muun muassa regressiotestaus, nopea palaute, nopeampi ajoaika manuaalitestaukseen verrattuna ja testaajien ajan vapautuminen suorittamaan muita testejä. He luettelevat myös haittoja, joita ovat testausautomaation tuoma väärä mielikuva laadusta, useista eri syistä epäonnistuvat testit, automaation ja testauksen sekoittamisen, testausautomaation ylläpidon vaatiman työmäärän sekä sen, että automaatiolla ei välttämättä löydetä paljon vikoja.

9 TESTAUSAUTOMAATIOJÄRJESTELMÄ

Testausautomaatiojärjestelmä (Test Framework) on kuvattu eri lähteissä hiukan eri tavoilla. Agarwalin (2016) mukaan testausutomaatiojärjestelmänä voidaan pitää protokollien, sääntöjen, standardien ja ohjeiden yhdistelmää, joiden yhdistämiseen ja seurantaan testauskehikko tarjoaa puitteet. Ghanakota (2012, 9) taas kuvailee testausautomaatiojärjestelmän helpommin ymmärrettävällä tavalla eli kokonaisvaltaisena automaattisten testien kehitys- ja ajoympäristönä, joka määrittää tiettyjä oletuksia, konsepteja ja käytäntöjä.

Testausautomaatiojärjestelmät voidaan jakaa ryhmiin testausutomaatiojärjestelmän toteutustavan ja siinä käytetyn tekniikan perusteella. Eri lähteissä ryhmäjaot ovat samanlaisia. Ghanakota (2012, 11) jakaa testausautomaatiojärjestelmät neljään eri ryhmään eli modulaarinen (Modular), datapohjainen (Data-Driven), avainsanapohjainen (Keyword-Driven) sekä edellisten yhdistelmä (Hybrid). Software Testing Help (2016) listaa edellisten neljän ryhmän lisäksi kirjasto- (Library Architecture) ja käytöspohjaiset (Behavior Driven) järjestelmät.

9.1 Vaihtoehtoisten työkalujen vertailu

SRAN/SBTS-järjestelmätason testausstrategian mukaan testausautomaatiojärjestelmänä tulisi käyttää jo ennestään käytössä olevaa työkalua ja mahdollisuuksien mukaan uudelleen käyttää myös olemassa olevia testitapauksia. Sisäisten keskustelujen jälkeen työkaluvertailuun valittiin kaksi vaihtoehtoa eli Flame ja Robot Framework. Eräänä vaihtoehtona olisi ollut myös Pegasus, joka on aikoinaan silloisen Nokia Siemens Networksin sisäiseen käyttöön kehitetty, Eclipse RCP- (Rich Client Platform) alustalle rakennettu testaustyökalu (Mazurkiewicz, M. 2010, 29). Se jätettiin vertailusta kuitenkin pois, sillä tässä projektissa mukana oleville henkilöille kyseisestä työkalusta ei ollut aikaisempaa kokemusta ja toisaalta sen käyttö oli saatavilla olleiden tietojen mukaan hiipumassa.

Flame on Nokian sisällä kehitetty ja käytössä oleva testausautomaatiojärjestelmä, joka alun perin kehitettiin WCDMA-tukiasemien testausautomaatioon. Flamea käytetään edelleen erilaisiin WCDMA- ja LTE-tukiasemien sekä RF Sharing -konfiguraatioiden automaattisiin testauksiin. Fla-

men alkuperäisenä ideana on ollut ohjata perinteistä WCDMA-tukiasemaa sen LMP- (Local Management Port) liitynnän kautta ilman tarvetta käyttää tukiaseman erillistä BTS Manager -sovellusta sen graafisine käyttöliittymineen. Flamen ydin on kirjoitettu C# -kielellä ja käyttäjän testitapaukset kirjoitetaan Python-kielisinä skripteinä. Tässä yhteydessä olennaisin seikka on se, että Flame on alun perin tehty nimenomaan BTS- eli tukiasematason testaukseen.

Robot Framework on Apache 2.0 -lisenssiin pohjautuva avoimen lähdekoodin yleiskäyttöinen testausautomaatiojärjestelmä, jonka käyttö perustuu avainsanoihin. Se on lisäksi sovellus- ja teknologiariippumaton, mikä tekee siitä hyvin yleiskäyttöisen. Nokian sisällä sitä käytetään jo entuudestaan muun muassa NetActin testaukseen. Robot Frameworkin ydin on kirjoitettu Pythonilla ja siihen on helppo integroida itse tehtyjä Python-kielellä toteutettuja osia. (Robot Framework 2016, viitattu 28.11.2016.)

Sisäisissä keskusteluissa Flamesta käytetään usein myös nimitystä ”testing framework” testausautomaatiojärjestelmän (Test Framework tai Test Automation Framework) sijaan. Tällä pyritään kuvaamaan paremmin Flamen käyttöä nimenomaan itse testaukseen. Flamen voisi nähdä olevan astetta lähempänä itse testattavaa järjestelmää kuin Robot Framework. Tämän takia Flamen lokerointi johonkin edellä mainituista testausautomaatiojärjestelmien ryhmistä ei tuntunut täysin luontevalta, mutta se lienee lähinnä modulaarista vaihtoehtoa. Taulukossa 2 on vertailtu tämän opinnäytetyön kannalta keskeisimpiä Flamen ja Robot Frameworkin ominaisuuksia.

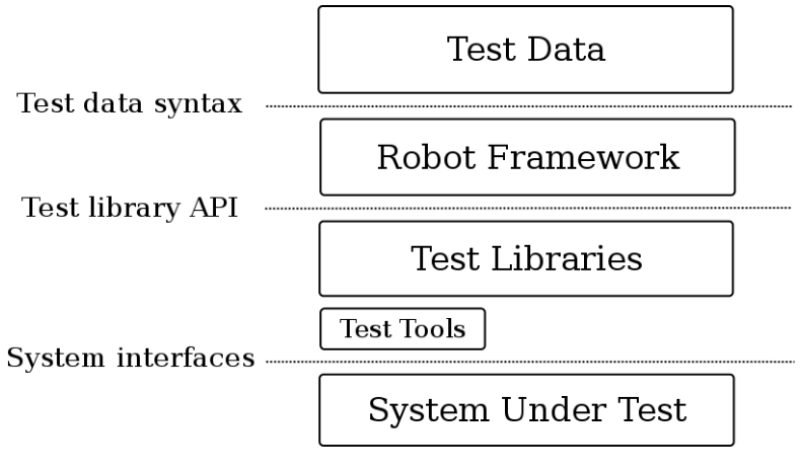
TAULUKKO 2. Eräiden Flamen ja Robot Frameworkin keskeisten ominaisuuksien vertailu

Vertailun kohde	Flame	Robot Framework
Nykyinen käyttöalue	BTS testaus	NetAct testaus
Valmiita testitapauksia NetAct sovellusten testaukseen	Ei	Kyllä
Valmiita testitapauksia SBTS WebUI testaukseen	Ei	Ei
Puhelingeneraattorin ohjaus valmiina	Kyllä (MACG, MCG)	Ei (Tulossa)
Valmiit kirjastot ulkoisten mittalaitteiden ohjaukseen	Kyllä	Ei

Vertailun kohteeksi valittujen seikkojen perusteella työhön valittiin Robot Framework, jonka suurimpana etuna Flameen verrattuna nähtiin sen käyttö jo ennestään NetAct-testauksessa ja kyseistä testausta varten kehitetyt testikirjastot. Niitä voitiin käyttää joko sellaisenaan tai pohjana SBTS-testitapausten suunnittelussa. Suurimpana puutteena Robot Framework vaihtoehdossa oli projektin alussa puuttunut kirjasto puhelingeneraattoreiden ohjaukseen. Kirjasto oli kuitenkin jo kehitteillä, joten puutetta ei nähty käyttöönoton esteenä. Ulkoisten mittalaitteiden, kuten RF tehomittarin tai spektrianalysaattorin, käyttö järjestelmätason testauksessa arvioitiin tarpeelliseksi vasta myöhemmin, eikä sopivien testikirjastojen puutetta nähty sen vuoksi tässä vaiheessa ongelmaksi.

9.2 Robot Framework -järjestelmän arkkitehtuuri

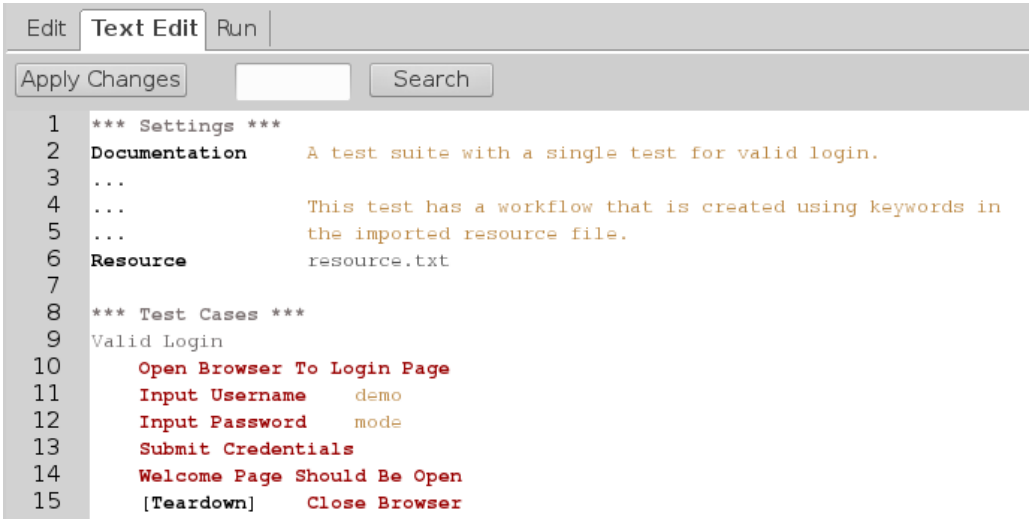
Robot Framework -järjestelmän modulaarinen arkkitehtuuri on esitetty kuvassa 17, jossa ylimmällä tasolla on testidata (Test Data). Se sisältää käyttäjän määrittämät avainsanat ja itse testitapauksen (test cases) toiminnot ja logiikan.



KUVA 17. Robot Frameworkin arkkitehtuuri (Robot Framework 2016)

Testitapauksia voi luoda Robot Frameworkille myös ulkoisilla tekstieditoreilla mutta tässä työssä käytettiin Robot Frameworkin omaa RIDE-editoria. Se tunnistaa Robot Frameworkin käyttämän syntaksin ja helpottaa näin testitapauksen kirjoittajan työtä. RIDE:ssä havaittiin myös muutamia ikäviä ominaisuuksia kuten vaikkapa äskettäin tehtyjen muutosten yhtäkkinen katoaminen jonkin testauskoodin syntaksiin eksyneen kirjoitusvirheen takia. Pari kertaa testitapaus hävisi RIDE:stä kokonaan, mutta testitapauksen sisältävä tiedosto saatiin palautettua muuta kautta. Näiden ominaisuuksien kanssa oppi kuitenkin elämään kokemuksen karttumisen myötä.

Kuvassa 18 näkyy tyypillisen testitapauksen rakenne, jossa avainsanat erottuvat punaisella kirjasinlajilla.



KUVA 18. Tyypillinen Robot Framework testitapaus

Robot Frameworkin varsinainen testauskyky perustuu kirjastoihin (Test Libraries). Robot Frameworkin ydin ei tiedä itse testattavasta sovelluksesta tai järjestelmästä vaan kommunikointi sen kanssa tapahtuu testikirjastojen kautta. Robot Frameworkin mukana tulee useita vakiokirjastoja, jotka sisältävät usein tarvittavia avainsanoja erilaisten tavanomaisten ohjelmallisten operaatioiden suorittamiseen. Näistä ovat esimerkkeinä vaikkapa erilaiset vertailut, silmukat ja tekstijonojen käsittelyt sekä vaikkapa Telnet-yhteyden käyttöön liittyvät tavanomaiset operaatiot ja niiden avainsanat.

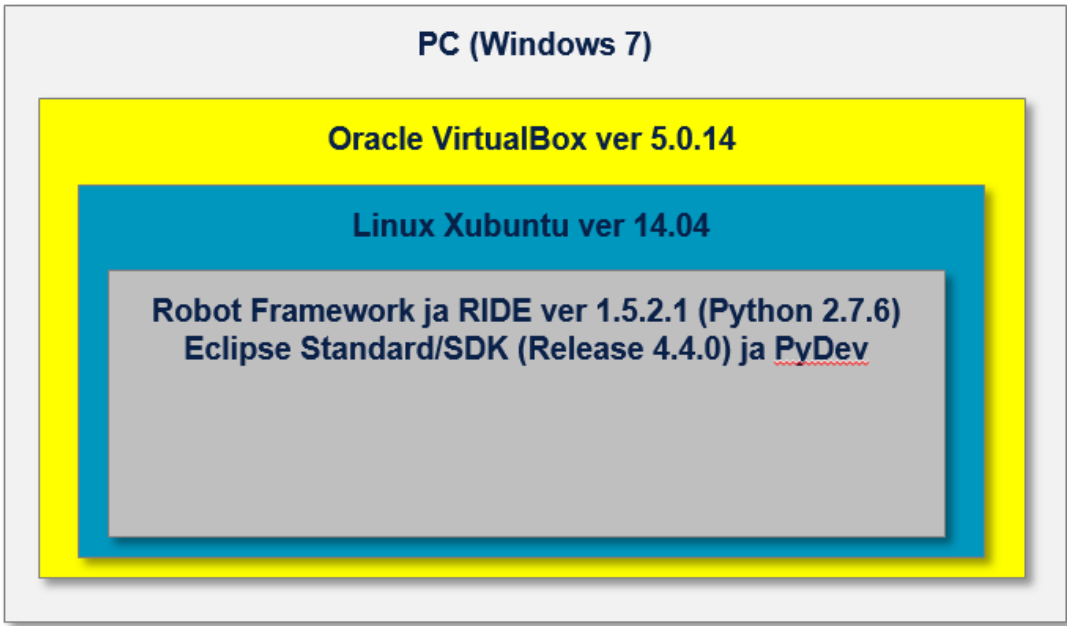
Kun Robot Framework käynnistetään, se prosessoi testidatan, ajaa testitapauksen ja luo testilokit ja -raportit. Selkeiden raporttien ja lokien automaattinen luonti havaittiinkin erääksi parhaista Robot Frameworkin tarjoamista ominaisuuksista testitapauksia kehitettäessä ja niitä ajettaessa.

10 KEHITYS- JA TUOTANTOYMPÄRISTÖN KUVAUS

10.1 Kehitysympäristö

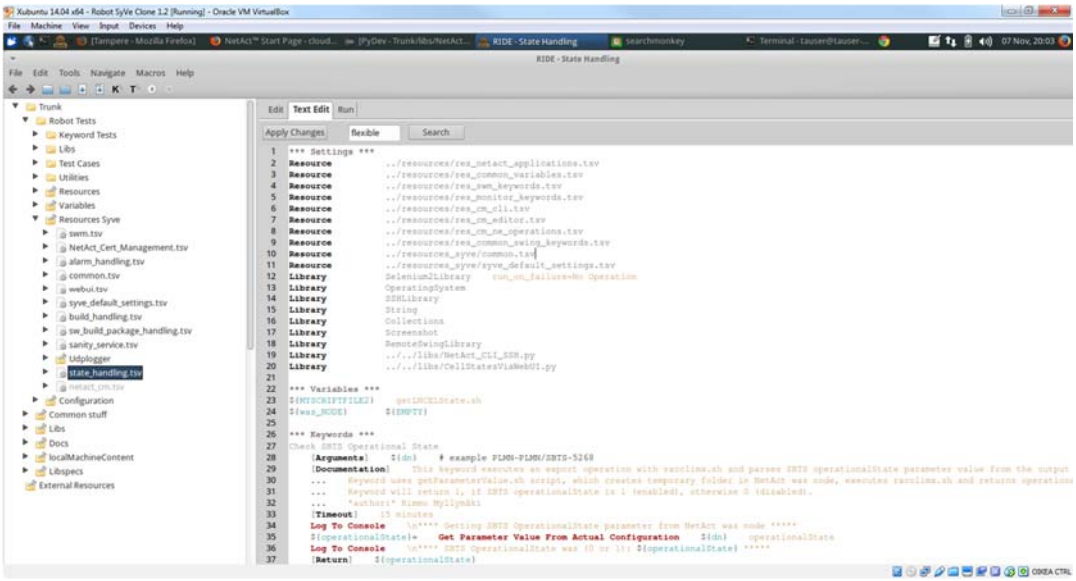
Kehitysympäristöllä tässä yhteydessä tarkoitetaan omalle tietokoneelle asennettuja ohjelmia, joilla pystytään kehittämään ja testaamaan erilaisia testitapauksia käyttäen Robot Frameworkia ja sen sisäistä RIDE-tekstinmuokkausohjelmaa.

Lopullisessa tuotantoympäristössä kehitettyjä testejä ajetaan Linux-käyttöjärjestelmällä toimivien palvelimien sisällä olevissa virtuaalikoneissa, minkä vuoksi myös kehitysvaiheessa Robot Frameworkia haluttiin käyttää Linuxin päällä. Tätä varten omaan Windows-käyttöjärjestelmällä toimivaan tietokoneeseen asennettiin ensin Oracle VirtualBox, joka on käyttöjärjestelmien ja muiden ohjelmien virtualisointiin tarkoitettu ilmainen ohjelma. VirtualBoxin sisään asennettiin Linux Xubuntu -jakelu ja sen päälle Robot Framework ja sen RIDE-editori, Eclipse-ohjelmointiympäristö ja sen Python-editori PyDev sekä muita tarvittavia apuohjelmia. Kehitysympäristön arkkitehtuuri on esitetty kuvassa 19.



KUVA 19. Kehitysympäristön arkkitehtuuri

Tyypillinen kuvakaappaus Robot Frameworkin RIDE-näkymästä VirtualBoxiin asennetussa kehitysympäristössä on esitetty kuvassa 20.

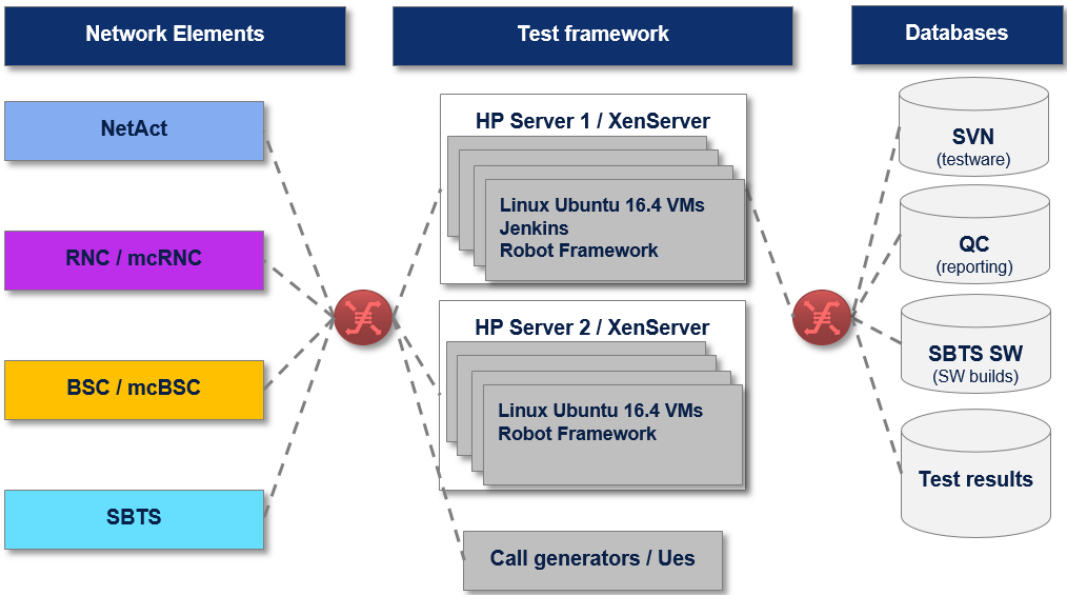


KUVA 20. Tyypillinen RIDE-näkymä

10.2 Tuotantoympäristö

Tuotantoympäristöllä tarkoitetaan tässä yhteydessä kokonaisuutta, johon kuuluvat testauksen kohteena olevat verkkoelementit (Network Elements), testaukseen käytettävät ohjelmistot ja niiden palvelimet (Test Framework) sekä erilaiset tietokannat (Databases), joissa joko säilytetään testauksessa tarvittavia tukiaseman ohjelmistoja tai niihin kirjoitetaan testituloksia. Tuotantoympäristö on erittäin laaja kokonaisuus ja siihen kuuluu lukuisa määrä erilaisia elementtejä.

Kuva 21 esittää tuotantoympäristön arkkitehtuuria.

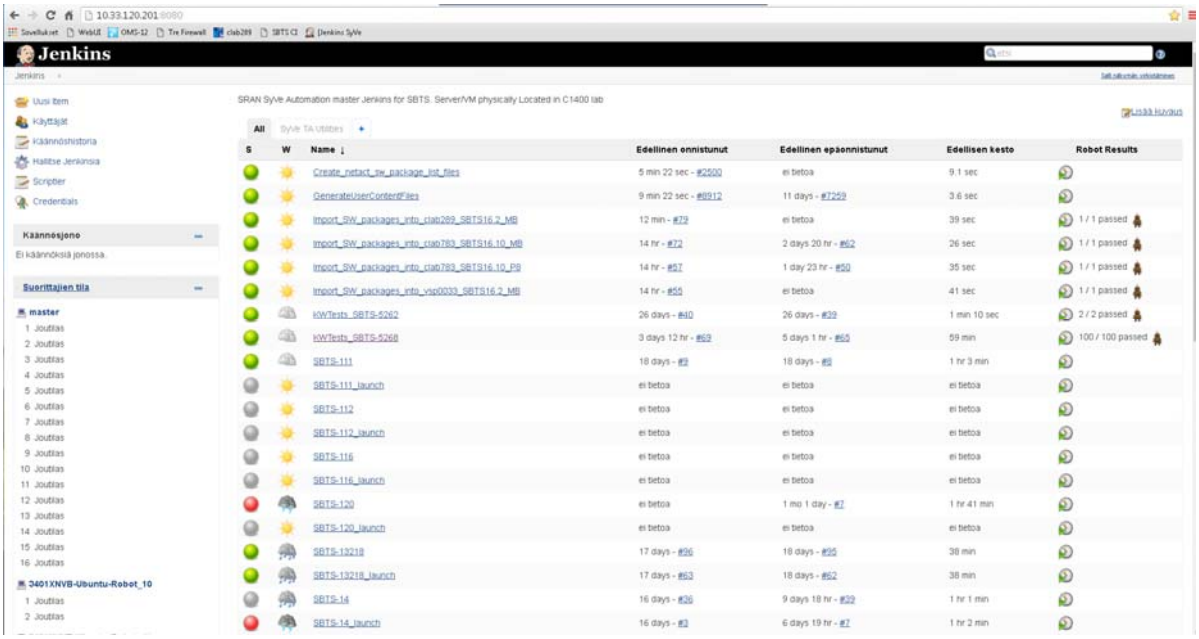


KUVA 21. Tuotantoympäristön arkkitehtuuri

Tuotantoympäristössä käytetään samoja verkkoelementtejä kuin kehitysympäristössä lukuun ottamatta vain tuotantotestaukseen nimettyjä SBTS-tukiasemia. Testiajoja voidaan suorittaa kahdessa HP:n palvelimessa, joihin on konfiguroitu Linux Ubuntu -käyttöjärjestelmää käyttäviä virtuaalikoneita. Yhteen virtuaalikoneeseen on asennettu tuotantoympäristön sydän eli Jenkins, joka on avoimeen lähdekoodiin perustuva automaatiopalvelin (Jenkins 2016, viitattu 28.11.2016). Sen avulla voidaan rakentaa erilaisia jatkuvassa integraatio- ja regressiotestauksessa tarvittavia testausrutiineja. Muihin virtuaalikoneisiin on asennettu Robot Framework tarvittavine lisäosineen varsinaisten testiajojen suoritusta varten. Kehitysympäristössä kehitetyt testikirjastot testitapauksineen talletetaan SVN-versionhallintajärjestelmään. Testattavassa SBTS-tukiasemassa käytettävät ohjelmistot haetaan tietokannasta. Testitulokset talletetaan tietokantaan ja verkkolevylle sekä testitulokset QC-tietokantaan.

Testiajo voidaan käynnistää Jenkinsissä automaattisesti halutuilla ehdoilla tai manuaalisesti valitsemalla testauksessa käytettävät parametrit ja SBTS-tukiasema. Ajon alussa Jenkins varaa yhden edellä mainituista virtuaalikoneista ja hakee SVN:stä uusimmat testikirjastot. Varsinaiset testit Jenkins ajaa käyttämällä virtuaalikoneeseen asennettua Robot Frameworkia. Tämän projektin yhteydessä Jenkinsiä käytetään pääasiassa regressiotestaukseen ajamaan joko CRT1- tai CRT2-testikokoelmien sisältämiä testitapauksia.

Kuvassa 22 on tyypillinen Jenkins-näkymä, josta käyttäjä voi käynnistää testiajoja.



KUVA 22. Tyypillinen Jenkins-näkymä

11 TESTITAPAUKSET

11.1 Testausohjelmiston uudelleenkäyttö

”Hyvät ohjelmoijat koodaavat, erinomaiset uudelleenkäyttävät” (Thomas 2009, viitattu 28.11.2016). Tämä pitää hyvin paikkansa, sillä kuka tahansa ohjelmiston kehittäjä etsii internetistä omaa työtään vastaavia toteutuksia ja aiheeseen liittyviä keskusteluita ja sama pätee tässäkin työssä.

Ohjelmistojen testaukseen käytettävä testausohjelmisto (testware) testitapauksineen on jo nimensäkin perusteella eräs ohjelmiston laji ja sen kehittäminen on käytännössä ohjelmointia. Robot Framework ei tee tässä suhteessa poikkeusta. Tämän työn yhteydessä käytettiin uudelleen NetAct-testaukseen jo aiemmin Robot Frameworkillä kehitettyjä testitapauksia ja -kirjastoja. Kyseessä oli selkeästi white-box -tapaus, sillä testausohjelmiston lähdekoodi oli käytettävissä. Osa uudelleenkäytetyistä testitapauksista oli teknisesti sopivia ja hyvin dokumentoituja eli niiden uudelleenkäytettävyys oli hyvä. Toisaalta osaa testitapauksista voitiin käyttää vain mallina ja tässä havaittiin todeksi edellä mainittu toteamus siitä, että uudelleenkäyttö edellyttää koodin hyvää tuntemusta (Ravichandran & Rothenberger 2003, viitattu 28.11.2016.). Varsinkin työn alkuvaiheessa valmista testitapausta joutui usein tutkimaan pitkään, ennen kuin sen sisältämien muiden avainsanojen toiminta, erilaisten muuttujien tarkoitus ja ohjelman logiikka avautuivat. Vasta tämän jälkeen siihen pystyi tekemään tarvittavia muutoksia oman testitapauksen tarpeiden mukaan.

Robot Frameworkiin voi liittää myös itse kehitettyjä ulkoisia kirjastoja. Ulkoisten kirjastojen käytöstä saatiin tämän työn yhteydessä hyviä kokemuksia. Muutamia erilaisissa testitapauksissa tarvittavia toimintoja toteutettiin Python-kielisiin funktioihin, jotka talletettiin .py-päätteiseen tiedostoon. Testitapauksessa tiedosto sisällytettiin (import) Robot Frameworkissä kehitettävään testitapaukseen ja näin Pythonin funktioiden toimintoja pystyi helposti käyttämään samaan tapaan kuin muitakin kirjastoissa jo valmiina olevia funktioita ja avainsanoja. Keskusteluissa kokeneempien Robot Framework -kehittäjien kanssa selvisi, että testitapauksen ydintoiminnot ja tarvittavat toiminnot kannattaakin yleensä koodata Pythonilla ja sitten kutsua niitä Robot Frameworkin puolelta.

11.2 SRAN/SBTS-tukiaseman olioiden tilat

Tyypillisessä mobiiliverkossa voi olla jopa kymmeniä tuhansia tukiasemia ja verkko-operaattori voi kohdistaa jonkin operaation suureen joukkoon yhtä aikaa. Esimerkkinä olkoon vaikkapa tukiaseman ohjelmiston päivitys, jolloin kohteena voi olla kerrallaan jopa tuhansia tukiasemia. Operaattori luonnollisesti olettaa ja edellyttää, että päivitys onnistuu kerralla halutulla tavalla. Näin massiivisen tukiasemamäärän rakentaminen Nokian testauslaboratorioihin ei kuitenkaan ole mahdollista vaan tietyn operaation luotettavuutta haetaan toistamalla testiä useita kertoja. Manuaalisesti tehtynä tällaiset useita toistoja vaativat testaukset ovat työläitä ja turhauttavia sekä toisaalta myös alttiita inhimillisille virheille. Testausautomaation avulla voidaan sen sijaan väsymättä toistaa edellä mainittua prosessia aina samalla tavalla parantaen näin myös itse testausprosessin luotettavuutta.

Tyypillinen järjestelmätason regressiotestitapaus onkin uuden SRAN/SBTS-tukiaseman ohjelmistoversion lataus, aktivointi sekä paluu (roll-back) entiseen versioon NetActin kautta. Testitapaus voidaan suorittaa manuaalisesti mutta testausautomaatiolla sitä voidaan toistaa useita kertoja prosessin toiminnan luotettavuuden arvioimiseksi. Ohjelmiston testauksessa on tärkeä varmistaa, että järjestelmä raportoi olioiden toiminnalliset tilat oikein eri tilanteissa. Tyypillinen esimerkki on tukiasemalle CRT-ajon yhteydessä vaikkapa uuden ohjelmistoversion käyttöönotossa tapahtuva uudelleenkäynnistys (reset), jonka jälkeen tukiaseman ja kaikkien sen soluolioiden tulee palata aktiiviseen tilaan ja oikean tilan tulee näkyä niin SBTS-tukiasemassa, tukiasemaohjaimissa kuin NetAct:ssä.

11.3 Olioiden toiminnallisten tilojen tarkistaminen

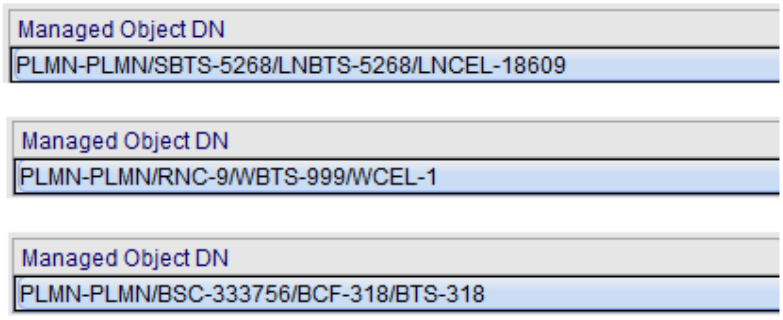
Olion tilanhallinnalla tarkoitetaan verkko-elementin kykyä raportoida sen eri olioiden tilat oikein ja toisaalta käyttäjän toimenpiteitä olioiden tilojen muuttamiseksi. Tiloja on kolmenlaisia eli toiminnallinen (operational), hallinnollinen (administrative) ja paikallinen (local). Olion toiminnallisen tilan asettaa järjestelmä automaattisesti kun taas hallinnollisen ja paikallisen tilan määrittää käyttäjä manuaalisesti. Olioiden raportoima toiminnallinen tila tunnetaan nimellä Operational State, joka voi olla joko Enabled (käytössä tai aktiivinen) tai Disabled (pois käytöstä tai passiivinen).

Tämän opinnäytetyön kannalta keskeinen osa on SRAN/SBTS-tukiaseman sisältämien olioiden toiminnallinen tila (Operational State), mitkä voi nähdä seuraavista paikoista:

- SBTS:n, LNBTS:ien ja LNCEL:ien toiminnalliset tilat voi nähdä NetAct Monitorin, NetAct Configuratorin ja SBTS:n WebUI:n kautta.
- WCEL:ien toiminnalliset tilat voi tarkistaa RNC:n ja NetAct Configuratorin kautta.
- BCF:n, BTS:ien ja TRX:ien toiminnalliset tilat voi tarkistaa BSC MML:n tai SBTS:n WebUI:n kautta.

(Nokia NetAct 16.2 Single RAN Management Overview 2016, 20.)

Toiminnallisten tilojen tarkistus NetActin kautta tapahtuu käyttämällä olion DN-tunnistetta viitteenä. Kuvassa 23 on esitelty tyypilliset LNCEL-, WCEL- ja BTS-olioiden DN-tunnisteet.



KUVA 23. Tyypilliset LNCEL-, WCEL- ja BTS-olioiden DN-tunnisteet

Kuvan yläreunassa oleva LTE-tukiaseman solu LNCEL-18609 sijaitsee LNBTS-5268 -olion alla, joka puolestaan sijoittuu SBTS-5268 -olion alle. Kuvassa alempana olevien WCDMA- ja GSM-tukiasemien solut WCEL-1 ja BTS-318 sijoittuvat vastaavasti 3G- ja 2G-tukiasemaa kuvaavien olioiden WBTS-999 ja BCF-318 alle ja ne edelleen tukiasemaohjaimien, RNC-9 ja BSC-33756, alle.

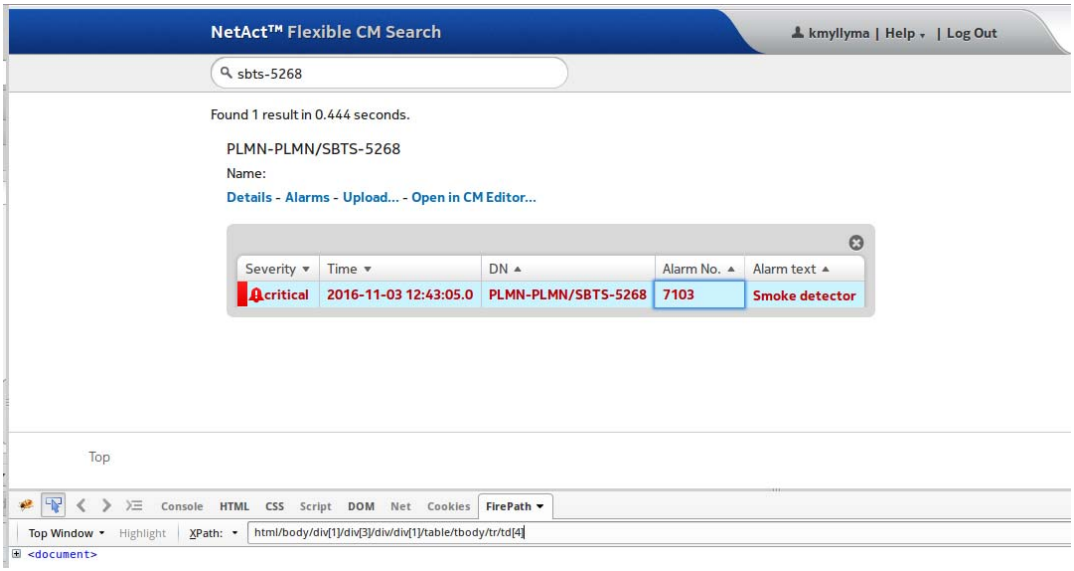
Kuten aiemmassa SRAN/SBTS-arkkitehtuuria kuvaavassa kappaleessa esiteltiin, on 3G- ja 2G-soluolioilla vain sukulaisuussuhde SBTS-olioon eikä niiden yksilöllistä DN-tunnistetta voi selvittää suoraan SBTS-olion DN-tunnisteen kautta. Sukulaisuussuhteen ja soluolioiden toiminnalliset tilat voidaan selvittää joko CM Editor -sovelluksen kautta tai käyttämällä NetActin sisäänrakennettuja CLI- (racclimx.sh ja ractoolsmx.sh) työkaluja.

LTE-, 3G- ja 2G-solujen toiminnalliset tilat voi tarkistaa myös SBTS-tukiaseman kautta eikä DN-tunnistetta tarvitse silloin käyttää. Tilojen tarkistukseen on tällöin kolme erilaista teknistä toteutustapaa: Joko suoraan verkkoselaimen graafisen WebUI-näkymän kautta, WebUI-näkymän muodostavan HTML-kuvauskielen kautta tai tukiaseman sisäisen BIM- (Base station Information Model) tietorakenteen kautta.

Testausautomaatiossa käytettävien tekniikoiden, työkalujen sekä testauskattavuuden kannalta erilaisten toteutustapojen erot ovat merkittävä. Seuraavassa tarkastellaan lähemmin eri teknisiä toteutustapoja.

11.3.1 NetAct - HTML

Flexible CM Search on eräs NetAct HTML -sovellus, jota käytetään verkkoselaimen kautta. Testausautomaation kannalta verkkoselaimen ikkunan sisältöä voi lukea Robot Framework -ympäristöön kuuluvien Selenium2Library-avainsanojen kautta. Kuvassa 24 on tyypillinen esimerkki, jossa käyttäjä on valinnut Flexible CM Search -ikkunassa näkyviin tietyn SBTS-tukiaseman aktiiviset hälytykset (Alarms).



KUVA 24. NetAct Flexible CM Search -näkymä

Verkkoselaimen HTML-näkymässä oleviin elementteihin voidaan viitata useilla eri tavoilla. Näitä ovat muun muassa elementin ID, nimi ja xpath (webperformance 2016, viitattu 28.11.2016). Halutun elementin xpath-polun voi selvittää esimerkiksi Firefox-selaimeen asennettavan Firebug-lisäosan avulla. Kuvan 24 esimerkissä näkyvän aktiivisen hälytyksen xpath-polku on Firebugin ikkunassa kuvan alareunassa ja sitä voi käyttää myös testausautomaatiossa hälytysten lukemiseen. Kuvan esimerkissä oleva xpath-polku on muotoa:

```
html/body/div[1]/div[3]/div/div[1]/table/tbody/tr/td[4].
```

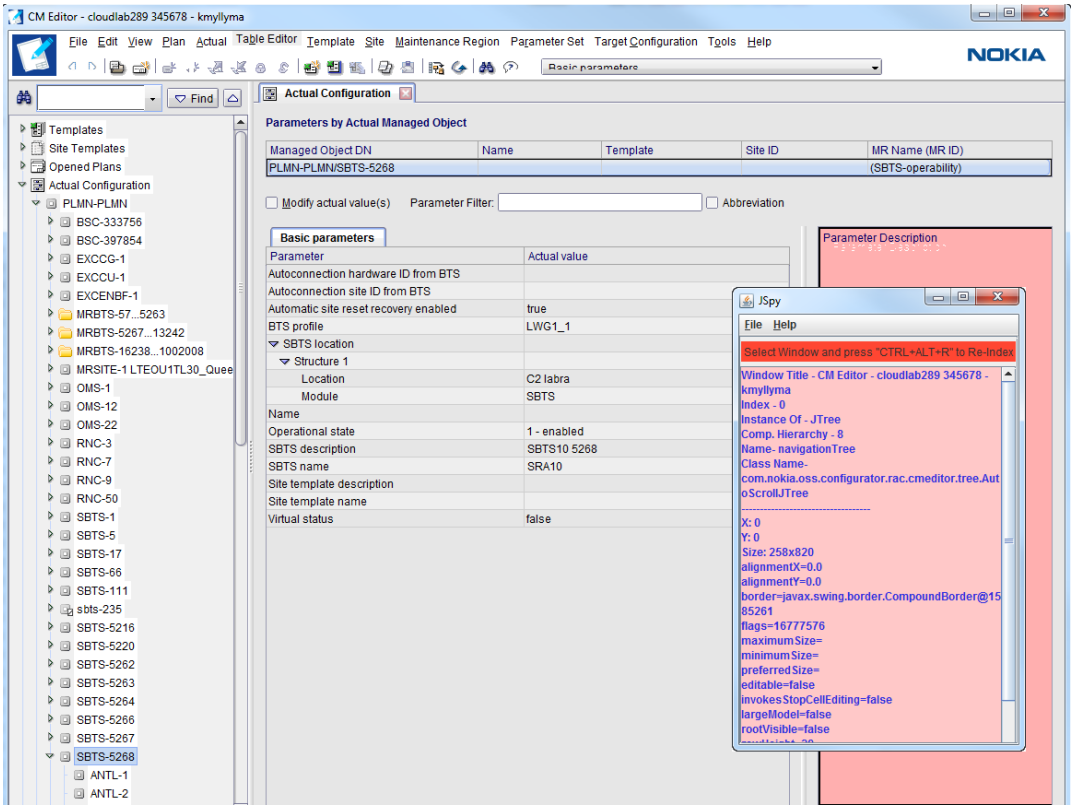
Jos elementin xpath-polku muuttuu testattavan sovelluksen kehittymisen myötä, tulee uusi polku lisätä myös testausautomaation koodiin.

Solujen toiminnallisia tiloja ei voi tarkistaa Flexible CM Search -sovelluksen kautta, joten tämä esimerkki on tässä yhteydessä vain mallina xpath-toteutuksesta ja siihen liittyvästä ylläpito-ongelmasta.

11.3.2 NetAct - Java

Eräs NetAct Java-sovellus on CM Editor, joka muiden NetAct-sovellusten tapaan käynnistetään NetActin pääsivulta. Sovelluksen sisältävä cmedit.jnlp-tiedosto latautuu ensin käyttäjän tietokoneelle ja sovellus avautuu sen jälkeen. Sovelluksen sisältämien valikoiden vaihtoehtoihin ja erilaista tietoa sisältävien kenttien sisältöön voi viitata Robot Frameworkin yhteydessä olevan Remote SwingLibrary -laajennuksen sisältämien avainsanojen kautta. Kenttien viittauksessa tarvittavat yksityiskohdat voi selvittää käynnistämällä CM Editor -sovelluksen JSpy-apusovelluksen kautta.

Kuvassa 25 on tyypillinen CM Editor -näkymä, jonka vasemmassa reunassa olevaan oliopuuhun voidaan viitata sille koodissa annetun nimen perusteella. Kuvan oikeassa reunassa olevan JSpy-apusovelluksen avulla selviää, että puun nimi CM Editorin sisällä on ”navigationTree” ja juuri sitä tulee käyttää testausautomaation koodissakin puuhun viitattaessa. Tämän tyyppisissä viittauksissa on tosin vastaava ongelma kuin aiemmin kuvatun xpath-polun kanssa, eli toteutus voi mennä rikki testattavan Java-sovelluksen kehittymisen myötä.



KUVA 25. Tyypillinen CM Editor –näkymä

11.3.3 NetAct - CLI

NetActin tiettyjä toimintoja voi ajaa myös NetActin virtuaalikoneiden komentorivin eli CLI:n kautta. Tässä yhteydessä tarvittavat tärkeimmät CLI-työkalut ja niiden käyttötarkoitukset on esitelty taulukossa 3.

TAULUKKO 3. Tärkeimmät NetAct komentorivityökalut

Komentorivityökalu	Käyttötarkoitus
racclimx.sh	Configuration Management
ractoolsmx.sh	Tietokysely CM tietokannasta

Alla on esimerkki NetActin virtuaalikoneen komentoriville annetusta racclimx.sh -komennosta, jolle annetaan parametrina halutun SBTS:n DN-tunniste ja komento palauttaa sen jälkeen SBTS:n sisältämien LNCEL-luokkien tiedot export.xml-tiedostona.

```
# racclimx.sh -op Import_Export -DN "PLMN-PLMN/SBTS-5268" -importExpor-
tOperation actualExport -fileName export.xml -classFilter "LNCEL"
```

Palautettua export.xml -tiedostoa on helppo käsitellä Robot Frameworkin ja NetActin virtuaalikoneen komentorivi- eli niin sanotun shell-skriptin avulla ja etsiä tiedostosta halutut tiedot. Testausautomaation toteutuksen kannalta komentorivitoteutus on kaikkein helpoin, sillä silloin ei tarvitse avata NetActin sovelluksia pääikkunan kautta lainkaan.

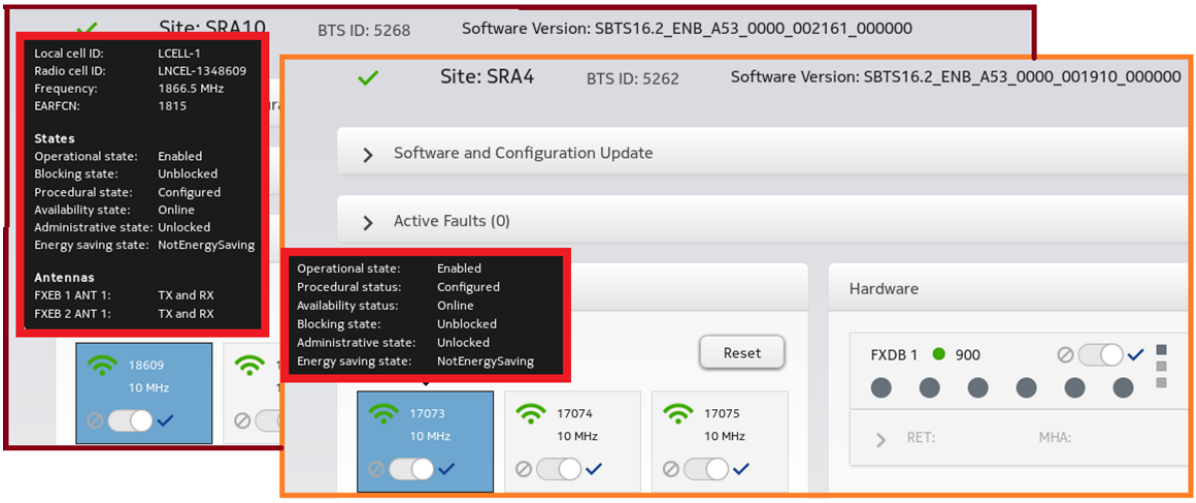
11.3.4 WebUI - GUI

Kuten aiemmin NetAct Flexible CM Search -tapauksessa, myös WebUI-näkymän sisältämiä elementtejä verkkoselaimessa voi lukea Robot Frameworkin Selenium2Library-avainsanojen avulla viittaamalla elementtien xpath-polkuihin. Tähän testitapaukseen kehitetty koodi klikkaa jokaista WebUI-näkymän solua ja lukee esiin tulevasta ponnahdusikkunasta kyseisen solun toiminnallisen tilan. Toiminnallisen tilatiedon sisältävän ponnahdusikkunan rivin xpath-polku kirjoitettiin testausautomaation koodiin.

Testitapausta kehitettäessä saatiin hyvä esimerkki siitä, kuinka helposti rikkoutuva menetelmä xpath-polun käyttö on. Kuvassa 26 on manuaalisesti klikattu yhtä solua kahdessa eri SBTS-tukiaseman WebUI-näkymässä. Tukiasemissa pyörii eri ohjelmistoversio siten, että uudempi ohjelmisto on käytössä takana olevassa WebUI-näkymässä. Sen solun tietoja sisältävässä mustapohjaisessa ponnahdusikkunassa on enemmän rivejä kuin etualalla olevassa toisen tukiaseman ponnahdusikkunassa. Testitapauksen kannalta olennainen Operational State -tieto on ponnahdusikkunoissa eri rivillä ja näin niiden xpath-polutkin ovat erilaisia. Ero vaikuttaa ihmissilmään pieneltä eikä aiheuta ongelmia manuaalisessa testauksessa, mutta koneellisen testausautomaation kannalta ero on merkittävä ja se tulee huomioida testitapauksen koodissa. Tässä tapauksessa ratkaisuna oli koodata molemmat mahdolliset xpath-polkuvaihtoehdot mukaan testitapauksen koodiin.

Myöhemmissä testeissä vielä uudemmilla SBTS-tukiaseman ohjelmistoilla havaittiin, että xpath-polut olivat jälleen kerran muuttuneet ja uudet polkuvaihtoehdot jouduttiin lisäämään testitapauksen

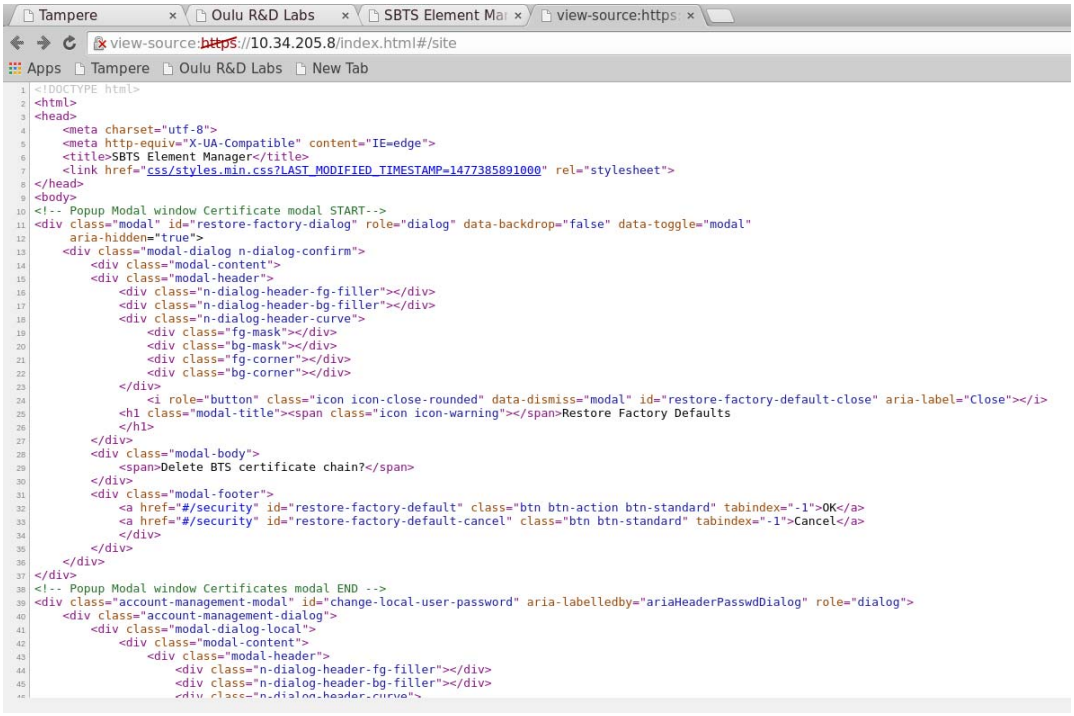
toteutukseen. Jos polku muuttuu vielä uudestaan myöhemmin, tulee testitapausta jälleen muokata vastaavasti.



KUVA 26. WebUI-näkymä kahdella eri ohjelmistoversiolla

11.3.5 WebUI - HTML

Verkkoselaimen ikkunaan avautuvan WebUI-näkymän muodostaa HTLM-koodi, jota voi halutessaan tarkastella verkkoselaimen kautta myös manuaalisesti. HTML-koodi sisältää myös solujen toiminnalliset tilat. Tämä testitapaus toteutettiin siten, että Robot Frameworkiin sisältyvällä Selenium2 -kirjaston Get Source -avainsanalla haettiin WebUI-näkymän HTML-koodi, joka sitten syötettiin itse tehtyyn Python-skriptiin. Skriptillä on helppo prosessoida koko koodi, etsiä sieltä halutut tiedot ja palauttaa haettu tieto takaisin itse testitapaukseen. Kuvassa 27 on tyypillinen esimerkki WebUI-näkymän takaa löytyvästä HTML-koodista.



KUVA 27. Tyypillistä WebUI-näkymän HTML-koodia

11.3.6 WebUI - BIM

BIM on SBTS:n OAM-ohjelmistolohkossa oleva tietorakenne, joka sisältää muun muassa systeemimoduuliin fyysisesti kytkettyjen ja tunnistettujen radiomoduuleiden tiedot sekä eri olioiden tilat. BIM-tietorakenteen voi hakea tukiasemasta tietokoneelle avoinna olevan WebUI-istunnon kautta http-tiedostohakuna (get) ja sen voi tallettaa .json-tyyppisenä tiedostona.

Tämän testitapauksen toteutuksessa haetun tiedoston käsittelyä varten suunniteltiin Python-skripti, joka etsii tiedostossa olevien soluolioiden toiminnallliset tilat.

11.3.7 BSC - CLI

Kuten aiemmin todettiin, GSM RAT:in sisältämien BCF-, BTS- ja TRX-olioiden toiminnalliset tilat voi tarkistaa BSC:ltä. Tarkistamista varten BSC:lle avataan Telnet-yhteys, jonka jälkeen BSC:n komentoriville annetaan MML-komento ZEEI. Tämän jälkeen BSC tulostaa parametrina annetun BCF-olion ja sen BSC:llä näkyvien BTS- ja TRX-olioiden toiminnalliset tilat. Tyypillinen ZEEI-tuloste

on esitetty kuvassa 28, jossa annetun BCF-318 -olion sisältämien muiden olioiden toiminnalliset tilat näkyvät ”OP STATE” -sarakkeessa.

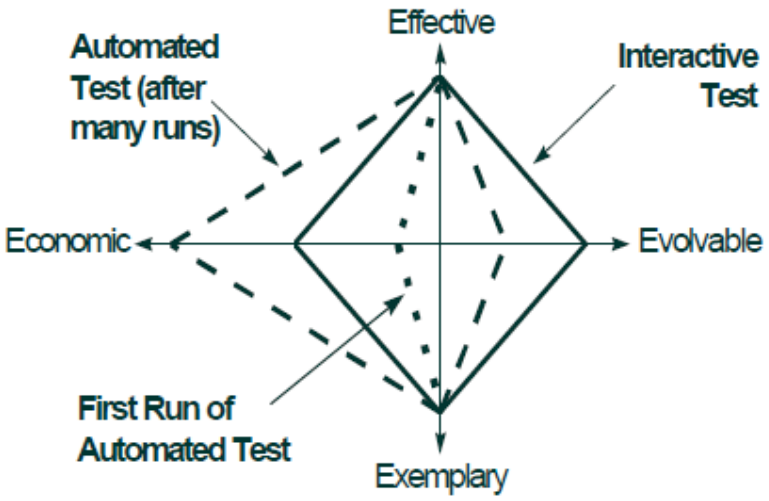
```
MAIN LEVEL COMMAND <___>
< ZEEI:BCF=318;

LOADING PROGRAM VERSION 36.3-0


FlexiBSC  SENJA                      2016-11-24  19:54:42
RADIO NETWORK CONFIGURATION IN BSC:
                                                    E P  B
                                          F         T R  C D-CHANNEL   BUSY
                        AD OP             R  ET-    R E  S O&M LINK   HR  FR
 LAC   CI         HOP ST STATE  FREQ      T  PCM ERACH      X F  U NAME   ST DHR DFR
                                                                          8K /GP
==================== == ====== ==== == ==== ========== = = == ===== == === ===
BCF-0318  FLEXI MR10   U WO     SBTS-5268                     1  OM318 WO
 00213 21318 BTS-0318  U WO                                                0   0
 SRA10BTS1       RF/-                                                      0   0
                                                                           0   1
              TRX-001  U WO       0  0    - MBCCHC         P  1
              TRX-002  U WO      92  0    -                   1
 00213 21319 BTS-0319  U WO                                                0   0
 SRA10BTS2       -/-                                                       0   0
                                                                           0   1
              TRX-003  U WO      40  0    - MBCCHC         P  1
              TRX-004  U WO      42  0    -                   1
 00213 21320 BTS-0320  U WO                                                0   0
 SRA10BTS3       RF/-                                                      0   0
                                                                           0   1
              TRX-005  U WO      50  0    - MBCCHC         P  1
              TRX-006  U WO      52  0    -                   1
 00213 21336 BTS-0336  U WO                                                0   0
 SRA10BTS4       -/-                                                       0   0
                                                                           0   1
              TRX-007  U WO     700  0    - MBCCHC         P  1
              TRX-008  U WO     702  0    -                   1
 00213 21337 BTS-0337  U WO                                                0   0
 SRA10BTS5       -/-                                                       0   0
                                                                           0   1
              TRX-009  U WO     710  0    - MBCCHC         P  1
              TRX-010  U WO     712  0    -                   1
 00213 21338 BTS-0338  U WO                                                0   0
 SRA10BTS6       -/-                                                       0   0
                                                                           0   1
              TRX-011  U WO     720  0    - MBCCHC         P  1
              TRX-012  U WO     722  0    -                   1

COMMAND EXECUTED
```

KUVA 28. Tyypillinen BSC:n ZEEI-tulostus

11.4 Testitapauksen hyvyys

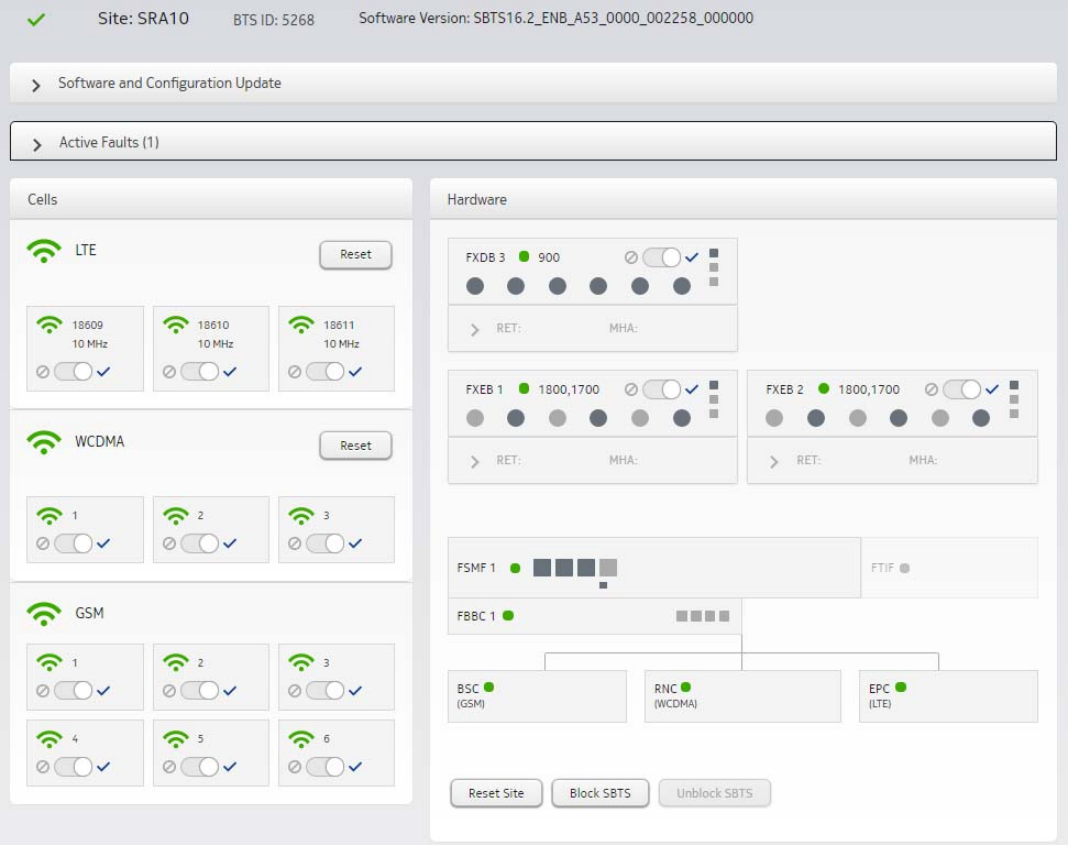
Testin tehokkuus voidaan määritellä sen perusteella, miten hyvin testillä löydetään vikoja. Tehokkuuden mittariksi voidaan asettaa testillä löydettyjen vikojen suhde kaikkiin löydettyihin vikoihin. (Hutcheson 2003, 120.)

Tässä yhteydessä tätä lähestymistapaa ei voitu kuitenkaan käyttää, sillä sekä SBTS-tukiaseman että testausautomaatio-ohjelmiston kehitys olivat vielä kesken eikä vikamääristä ollut lopullisia tietoja. Sen sijaan voidaan tarkastella testin hyvyyttä. Fewsterin (2001, viitattu 28.11.2016) mukaan testitapauksen hyvyyttä voidaan arvioida neljän ominaisuuden perusteella eli kuinka tehokas (Effective), kattava (Exemplary), taloudellinen (Economic) ja ylläpidettävä (Evolvable) testitapaus on. Tehokkuuden mittarina hän pitää sitä, kuinka monta ohjelmistovirhettä testitapaus löytää tai kuinka todennäköistä virheen löytyminen testitapauksen avulla on. Kattavuuden mittarina hän pitää sitä, kuinka montaa asiaa testitapaus testaa kerrallaan vähentäen näin tarvetta muihin testitapauksiin. Taloudellisuudessa hän huomioi testitapauksen suorituksen, analysoinnin sekä virheen etsinnän. Ylläpidettävyydellä hän tarkoittaa testitapauksen ylläpidon vaatimaa työmäärää, jos testattava ohjelmisto muuttuu. Näiden suhteen hän esittää tutkakaaviona kuvan 29 tapaan. Mitä suuremman pinta-alan muodostuva kuvio tekee, sitä parempana testitapausta voidaan pitää.



KUVA 29. Testitapauksen hyvyys (Fewster 2001, viitattu 28.11.2016)

Fewsterin esittämä malli testin tai testitapauksen hyvyyden arvioimiseksi ei kuitenkaan ota kantaa testitapauksen luotettavuuteen. Tässä tapauksessa myös se kuitenkin haluttiin selvittää sen arvioimiseksi, kuinka hyvin testitapauksen antamaan hyväksyttyyn (Pass) tai hylättyyn (Fail) tulokseen voi luottaa. Luotettavuutta ja keskimääräistä ajoaikaa mitattiinkin ajamalla tämän työn yhteydessä kehitettyjä testitapauksia 100 kierrosta Jenkins-tuotantoympäristössä käyttäen testauskohteena SBTS-tukiasemaa, jonka kaikki soluoliot olivat jatkuvasti aktiivisessa (Enabled) tilassa. Testeissä käytetyn SBTS-tukiaseman WebUI-näkymä on esitetty kuvassa 30.



KUVA 30. Testeissä käytetyn SBTS-tukiaseman WebUI-näkymä

Kaikki testitapaukset tuottivat odotetun hyväksytyn testituloksen jokaisella 100 kierroksella lukuunottamatta Check WCEL Operational States from CM Editor -testitapausta. Myös tätä testiä yritettiin ajaa tuotantoympäristössä 100 kierrosta, mutta se antoi väärän hylätyn tuloksen aina muutaman kymmenen kierroksen jälkeen eri syistä johtuen. Testausautomaation koodirivimäärällä mitattuna se on GUI-toteutuksena selkeästi muita toteutuksia pidempi ja monimutkaisempi, mikä näkyy myös sen keskimääräisessä ajoajassa. Kuten aiemmin todettiin, kaikki GUI-toteutukset ovat myös herkkiä menemään rikki testattavan kohteen muuttuessa. Kaiken tämän perusteella tätä testitapausta ei voi suositella käytettäväksi tuotantoympäristössä sen mahdollisesti antamien väärien tulosten vuoksi.

Taulukkoon 4 on merkitty toteutettujen testitapausten nimet, niiden tekniset toteutustavat ja yhden ajon keskimääräinen ajoaika. Kehitettyjen testitapausten tehokkuutta arvioitiin Fewsterin esittämän mallin avulla sitä hiukan soveltaen. Alkuperäisen mallin mukaan tässä arvioinnissa käytettiin neljää

mittaria ja kullekin niistä annettiin subjektiivisen näkemyksen mukaan arvoksi 1, 2 tai 3. Mitä suurempi luku, sitä nopeampi tai kattavampi testitapaus on tai sitä edullisempi se on kehittää tai ylläpitää.

Tehokkuuden mittariksi valittiin testitapauksen suoritusaika, joka taulukon 4 mukaisesti vaihtelee toteutustavasta riippuen 17–670 sekunnin välillä. Ajoajat kategorisoitiin arvoiksi 1, 2 ja 3 siten, että lyhimmän suoritusajan testit saivat arvoksi 3 ja pisimmän suoritusajan testi arvon 1.

TAULUKKO 4. Toteutettujen testitapauksien nimi, tekninen toteutustapa ja keskimääräinen ajoaika

Testitapauksen nimi	CLI/GUI	Tekninen toteutustapa	Ajoaika
Check SBTS Operational State	CLI	racclimx.sh + shell-skripti	17 s
Check LNCEL Operational State	CLI	racclimx.sh + shell-skripti	26 s
Check Cell Operational States from WebUI GUI	GUI	Selenium2Library	119 s
Check Cell Operational States from WebUI HTML	CLI	Selenium2Library + Python	25 s
Check Cell Operational States from WebUI BIM	CLI	Selenium2Library	28 s
Check WCEL Operational States from CM Editor	GUI	RemoteSwingLibrary	670 s [1]
Check WCEL Operational States from NetAct CLI	CLI	ractoolsmx.sh + Python	36 s
Check Operational States from BSC MML	CLI	Telnet + Python	23 s
Get WBTS DN from CM Editor [2]	GUI	RemoteSwingLibrary	

[1] Keskimääräinen ajoaika 40 kierroksen perusteella.

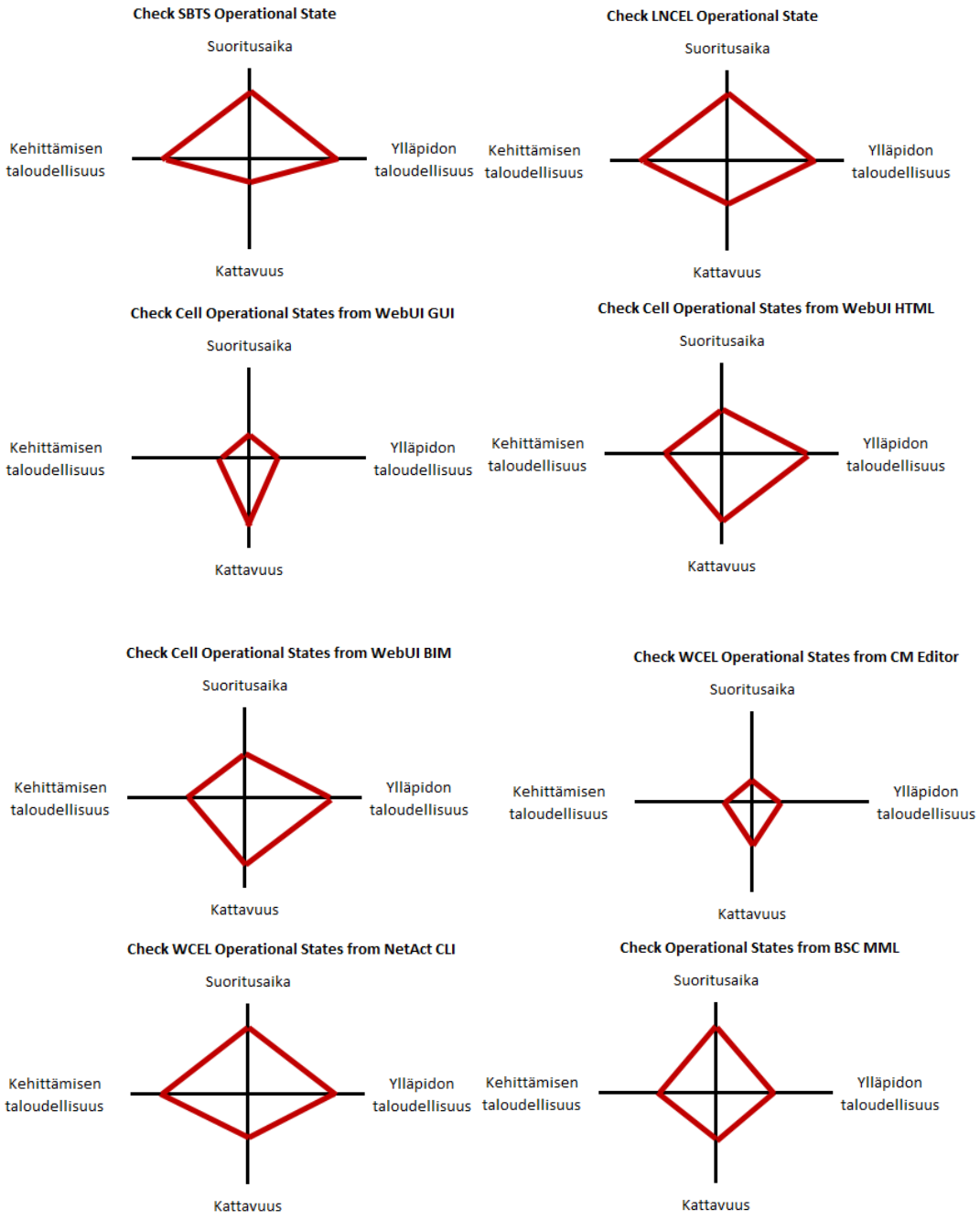
[2] Ajetaan automaattisesti Check WCEL Operational States from CM Editor -avainsanan yhteydessä.

Testitapauksien kattavuutta arvioitiin asteikolla 1–3 sen mukaan, kuinka montaa erityyppistä oliota kukin testitapaus kattaa kerrallaan. Kehittämisen ja ylläpidon taloudellisuutta arvioitiin samoin asteikolla 1-3. Yhteenveto testitapauksien kattavuudesta sekä kehittämisen ja ylläpidon taloudellisuudesta subjektiivisine arvioineen on koottu taulukkoon 5.

TAULUKKO 5. Yhteenveto testitapauksien kattavuudesta sekä kehittämisen ja ylläpidon taloudellisuudesta

Testitapauksen nimi	CLI / GLI	Testattavat oliot / Kattavuus	Kehittämisen taloudellisuus	Ylläpidon taloudellisuus
Check SBTS Operational State	CLI	SBTS	3	3
Check LNCEL Operational State	CLI	LNCEL	3	3
Check Cell Operational States from WebUI GUI	GUI	LNCEL, WCEL, BTS	1	1
Check Cell Operational States from WebUI HTML	CLI	LNCEL, WCEL, BTS	2	3
Check Cell Operational States from WebUI BIM	CLI	LNCEL, WCEL, BTS	2	3
Check WCEL Operational States from CM Editor	GUI	WCEL	1	1
Check WCEL Operational States from NetAct CLI	CLI	WCEL	3	3
Check Operational States from BSC MML	CLI	BCF, BTS, TRX	3	3

Taulukkojen 4 ja 5 perusteella piirrettiin tutkakaaviot, joissa pinta-alaerot tulevat selvästi näkyviin. Tutkakaaviot on esitelty kuvassa 31.



KUVA 31. Testitapausten hyvyyttä kuvaavat tutkakaaviot

Tutkakaavioiden perusteella on helppo havaita, että pinta-alan perusteella parhaita testitapauksia ovat CLI-pohjaiset toteutukset ja toisaalta huonoimpia GUI-tyyppiset ratkaisut. Havainto on täysin linjassa aiemmin esitettyjen teorioiden kanssa.

Toisaalta pelkästään testauskattavuuden kannalta katsottuna GUI-toteutukset ovat parhaita vaikkakin niiden tutkakuvioiden pinta-ala jää silti pieneksi. Tätä puutetta käytännössä kompensoi se,

että manuaalista testausta tehdään jatkossakin testausautomaation rinnalla ja silloin GUI:n kautta esille tulevat virheetkin on mahdollista havaita.

11.5 Testitapausten vertailu manuaaliseen testaukseen

Aiemmin luvussa 8.4 esitettiin testausautomaation tuomia etuja ja toisaalta haittoja. Tärkeimpinä etuina siellä lueteltiin luotettavan järjestelmän tuottaminen, testauskattavuuden lisääntyminen sekä testauksen työmäärän pieneneminen ja ajan säästö. Samoin automaattinen testaustulosten raportointi, tehokkaampi yhteensopivuus- ja regressiotestaus, pitkästyttävien ja arkipäiväisten testien ajo, ympärivuorokautinen testaus sekä testaajien ajan vapautuminen suorittamaan muita testejä nähtiin etuina. Haittoina taas lueteltiin testausautomaation tuoma väärä mielikuva laadusta, useista eri syistä epäonnistuvat testit, automaation ja testauksen sekoittamisen, testausautomaation ylläpidon vaatiman työmäärän sekä sen, että automaatiolla ei välttämättä löydetä paljon vikoja.

Tässä yhteydessä toteutetut testitapaukset liittyvät SBTS-tukiaseman olioiden toiminnallisiin tiloihin. Tilojen oikeellisuuden tarkistaminen on erittäin tärkeää kaikkien regressiotestiajojen yhteydessä, sekä ennen testiajojen käynnistämistä että jokaisen testikierroksen aikana. Sisäisissä keskusteluissa todettiin, että toteutettujen testitapausten vertailu manuaaliseen testaukseen ei ole järkevää. Manuaalinen testaus ei kaikkien testitapausten kohdalla ole käytännössä vaihtoehto testausautomaatiolle ketterään ohjelmistokehitykseen olennaisesti kuuluvassa regressiotestauksessa ja siihen liittyvässä jatkuvassa testitapausten toistossa. Toiminnallisten tilojen tarkistaminen eri paikoista manuaalisesti jokaisella testikierroksella olisi epätarkoituksenmukaista ja altis inhimillisille virheille. Ylipäätään CRT1- ja CRT2 -tyyppisten regressioajojen suorittaminen manuaalisesti vuorokauden ympäri ei olisi tarkoituksenmukaista eikä taloudellisestikaan järkevää. Testausautomaatiolla ajetut testitapaukset ovatkin näissä tapauksissa käytännössä ainoa vaihtoehto.

Tätä työtä kirjoitettaessa tuotantoympäristön käyttöönotto oli vielä menossa, minkä vuoksi lopullisia tuloksia eduista ja haitoista ei ollut käytettävissä.

12 POHDINTA

12.1 Työn tavoitteiden toteutuminen

Opinnäytetyön tavoitteena oli toteuttaa SBTS-tukiaseman olioiden toiminnallisten tilojen tarkistamiseen testitapauksia erilaisilla teknisillä ratkaisuilla, vertailla ratkaisujen ominaisuuksia ja tuloksia keskenään sekä arvioida testausautomaatiolla saavutettavaa hyötyä manuaaliseen testaukseen verrattuna toteutettujen testitapausten osalta.

Testausautomaatiojärjestelmäksi valittiin Robot Framework, jonka avulla kehitettiin seitsemän tekniseltä ratkaisultaan erilaista testitapausta. Osassa niistä käytettiin GUI-tyyppistä lähestymistä ja osassa taas CLI-ratkaisuja. Testitapauksien hyvyyttä verrattiin käyttämällä Fewsterin (2001, viitattu 28.11.2016) esittämää mallia sitä hiukan soveltaen. Vertailun perusteella CLI-ratkaisut olivat parhaita sekä kehittämisen että ylläpidon taloudellisuuden perusteella. Toisaalta testauskattavuuden kannalta GUI-ratkaisut todettiin parhaiksi. GUI-tyyppisissä ratkaisuissa käytettiin xpath-polkuja graafisten elementtien lukemiseen, mikä todettiin jo testitapausten kehitysvaiheessa helposti rikkoutuvaksi menetelmäksi. Havainto on linjassa Francinon (2015) artikkelin kanssa. Testausautomaatiossa tulisikin käyttää yksilöllisiä tunnisteita elementtien tunnistamiseen, mikä tulee huomioda jo testattavan ohjelmiston koodia suunniteltaessa (Webperformance 2016, viitattu 28.11.2016).

Työ liittyy järjestelmätason e2e-testaukseen, joka jo luonteeltaan edellyttäisi myös GUI:n kautta testaamista. Myös verkko-operaattorit käyttävät sekä GUI- että CLI-työkaluja. Molemmista seikoista johtuen GUI-ratkaisut puoltaisivat paikkaansa myös testausautomaatiossa. Automatisoitujen testien rinnalla suoritetaan kuitenkin aina myös manuaalisia testejä GUI:n kautta, joten täysin yksiselitteistä vastausta GUI:n käytön hyödyllisyydestä testausautomaatiossa on tämän työn kokemusten perusteella vaikea antaa.

Työssä todettiin, että toteutettujen testitapausten vertailu manuaaliseen testaukseen ei ole järkevää, sillä toteutettujen testitapausten ajaminen manuaalisesti ei käytännössä olisi mahdollista. Dustin, Rashka ja Paul (1999, 37–51) viittaavat Quality Assurance Instituten tekemään analyysiin, jonka mukaan automaattinen testaus vaatii vain 25% siitä työmäärästä, jonka sama testaus vaatii manuaalisesti. Analyysissä tarkastellaan tosin lähinnä itse testauksen vaatimaa aikaa ja työmäärää

huomioimatta testausautomaation kehittämiseen vaadittavaa aikaa. Kuten tämänkin työn yhteydessä havaittiin, juuri kehittämiseen ja ylläpitoon vaadittava aika ja työmäärä voivat olla huomattavan suuria erityisesti silloin, jos käytettävä testausautomaatiotyökalu ja testattava tuote ovat kehittäjille uusia ja jos testattava kohde kehittyy koko ajan. Kehitys- ja tuotantoympäristöjen luominen ja niiden ylläpito ovat myös vaativia ja työläitä vaiheita erityisesti silloin, jos testausautomaatioprojekti aloitetaan tyhjästä.

Kehitettyjä testitapauksia voidaan käyttää sekä testausautomaatiojärjestelmän alkutilanteen tarkistuksessa että jatkuvissa CRT-ajoissa.

12.2 Muita kokemuksia ja havaintoja

Testitapauksia kehitettäessä havaittiin, että Robot Frameworkin ja sen sisältämien kirjastojen dokumentaatio oli varsin suppea sisältäen vain olennaisimmat tiedot jonkin sisäänrakennetun avainsanan tarvitsemista argumenteista ja avainsanan palauttamasta arvosta. Internetistä ei myöskään löytynyt Robot Frameworkin kirjastojen avainsanoilla toteutettuja käytännön testitapauksia kovin paljon malliksi. Tämä aiheutti ongelmia ja uuden oppiminen vei aikaa erityisesti työn alussa. Kokemuksen lisääntymisen myötä suppeallakin dokumentoinnilla päästiin toki eteenpäin.

Työssä uudelleen käytettiin NetActin testaukseen jo aiemmin kehitettyjä testitapauksia. Työssä havaittiin selvästi, että osa valmiista testitapauksista oli helpommin uudelleenkäytettäviä kuin toiset. Vaikka alkuperäinen testauskoodi oli käytettävissä eli kyseessä oli white-box -tilanne, vei jonkun toisen tekemän testauskoodin ymmärtäminen joskus paljonkin aikaa. Alkuperäisen koodin logiikan ymmärtäminen oli kuitenkin edellytys sille, että koodia pystyi muokkaamaan omiin tarpeisiin sopivaksi. Tämä havainto oli täysin linjassa Ravichandranin ja Rothenbergerin (2003, viitattu 28.11.2016) kirjoittaman artikkelin kanssa.

Työn yhteydessä havaittiin myös, että testausautomaatio ei voi koskaan korvata kokenutta testaajaa, jolle saattaa testauksen edetessä tulla mieleen suorittaa jokin tietty testitapaus hiukan toisella tavalla kuin aiemmin. Kokenut testaaja saattaa myös manuaalista testausta tehdessään keksiä kokonaan uuden testitapauksen. Eräänä käytännön haasteena onkin saada nämä ideat mukaan myös testausautomaation koodiin. Manuaalista testausta tekevän henkilön pitäisi pystyä vähintäänkin määrittelemään uusi näkökulma tai testitapaus siten, että testausautomaatiota kehittävä

henkilö voi sen perusteella muuttaa olemassa olevaa koodia tai luoda uuden testitapauksen. Eräs vaihtoehto voisi olla sekin, että manuaalista testausta tekevä henkilö pystyisi itse muokkaamaan olemassa olevia testitapauksia testauskattavuuden lisäämiseksi. Näiden havaintojen perusteella ei mielestäni ole nähtävissä ainakaan lähitulevaisuudessa, että testausautomaatio voisi korvata ihmisen suorittaman manuaalisen testauksen. Testausautomaatiolla on kuitenkin selkeät etunsa. Näistä ehkä tärkeimpänä on mahdollisuus ajaa testejä jatkuvasti, jolla parannetaan sekä testauskattavuutta että testausautomaatiokäyttöön investoitujen laitteistojen käyttöastetta. Testausautomaatiolla voi ajaa samaa testiä lukemattomia kertoja, mikä manuaalisesti tehtynä olisi todella väsyttävää ja inhimillisille virheille altista. Jatkossakin on siis tarvetta sekä manuaaliselle että automaattiselle testaukselle.

Mobiiliverkko, järjestelmätason testaus ja testausautomaatio ovat aiheina erittäin laajoja, joten oleellisen aineiston valitseminen tämän työn teoriaosaan sekä kehitettävän työn näkökulman löytäminen olivat suurimpia haasteita. Näkökulmaksi muotoutui lopulta toiminnallisten tilojen tarkistaminen erilaisilla teknisillä ratkaisuilla, joka osoittautui hyväksi aiheeksi. Yhtäältä se antoi mahdollisuuden opiskella itselle entuudestaan vähemmän tuttuja asioita. Toisaalta testausautomaation liittyvien kehitys- ja tuotantoympäristöjen ja erilaisten teknisten ratkaisujen tultua tutuiksi voi opittuja asioita hyödyntää testausympäristöjen sekä uusien testitapausten kehittämisessä.

Eräänä mielenkiintoisena haasteena oli myös suomeksi kirjoittamisen vaikeus. Aiheeseen liittyvä aineisto on usein englanniksi ja osa englanninkielisestä terminologiasta on vakiintunut myös tekniseen puhekieleen. Tästä hyvänä esimerkkinä vaikkapa tässä työssä keskeinen sana object, josta omien havaintojeni mukaan suomenkielisessä puheessa käytetään sanaa objekti. Tässä työssä sen suomenkielisenä vastineena on kuitenkin käytetty sanaa olio, jota suositellaan käytettäväksi varsinkin tietojenkäsittelyn ja ohjelmoinnin yhteydessä (Sanakirja 2016, viitattu 28.11.2016).

12.3 Jatkokehitys

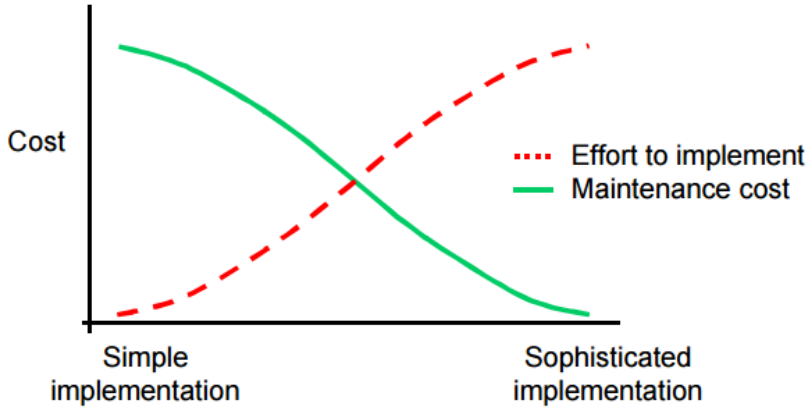
SRAN17-ohjelmisto tulee muuttamaan varsinkin SBTS-tukiaseman graafista käyttöliittymää. Testausautomaation tulee toistaiseksi tukea sekä nykyisiä SRAN16.2- ja SRAN16.10- että tulevaa SRAN17-ohjelmistoja, joten testausautomaatiokoodin ylläpito- ja kehitystyötä on edessä todella

paljon. Testauskattavuuden lisäämiseksi tuotantoympäristöön lisätään testipuhelut sekä spektrianalysaattori, jolla radiomoduuleiden lähetteen olemassaolon voi tarkistaa vastaavan soluolion tilan ollessa aktiivinen.

Työn aikana opittiin, että testausautomaation kehitys- ja tuotantoympäristöt ovat jo yksinään laajoja ja monimutkaisia kokonaisuuksia puhumattakaan testitapausten suunnittelusta ja koodaamisesta sekä radioverkon ja tukiasemien toiminnasta. Testausautomaatio jo sanana saattaa antaa väärän mielikuvan, sillä varsinkin järjestelmätason testausautomaation kehittäminen vaatii osaamista sekä testattavasta järjestelmästä, käytettävästä testausautomaatiotyökalusta että koodaamisesta. Omien kokemusteni perusteella onkin helppo yhtyä Kanerin (2000, 3) näkemykseen siitä, että pitäisikin mieluummin puhua tietokoneavusteisesta testauksesta.

Tässä työssä testauksen kohteena ovat SRAN/SBTS-tukiasema ja NetAct, jotka molemmat kehittyvät koko ajan. Työn kohteena oleva järjestelmätason testausautomaatio ei tämän vuoksi olekaan koskaan valmis vaan tulee vaatimaan jatkossakin osaavia resursseja niin uusien testitapausten kehittämiseen että entisten ylläpitoon. Havainto on linjassa myös Ghahrain (2015, viitattu 28.11.2016) mielipiteen kanssa. Fewsterin (2001, viitattu 28.11.2016) mukaan kasvavia ylläpitokuluja voi pyrkiä minimoimaan suunnittelemalla testitapaukset huolella jo alun perin, joka tosin nostaa kustannuksia testitapauksen luontivaiheessa. Riippuvuusmalli on esitetty kuvassa 32.

Testausautomaatiojärjestelmän ja testitapausten kehittäminen tulisikin nähdä investointina, joka maksaa itsensä takaisin myöhemmin.



KUVA 32. Testitapauksen ylläpitokustannusten riippuvuus testitapauksen toteutustavasta (Fewster 2001, viitattu 28.11.2016)

LÄHTEET

3gwiz. 2016. Networks. Viitattu 28.11.2016, http://3gwiz.com.au/ozmobilenet/?page_id=430.

Agarwal, D. 2106. Why do we need frameworks for test automation? Viitattu 28.11.2016, https://www.quora.com/Why-do-we-need-frameworks-for-test-automation.

Britton, T., Jeng, L., Carver, G., Cheark, P. & Katzenellenbogen, T. 2012. Reversible Debugging Software - "Quantify the time and cost saved using reversible debuggers". University of Cambridge. Viitattu 28.11.2016, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.370.9611.

Cohn, M. 2009. The Forgotten Layer of the Test Automation Pyramid. Viitattu 28.11.2016, https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

Dustin, E., Garrett T. & Gauf B. 2009. Implementing Automated Software Testing. Reading, Massachusetts. Pearson Education, Inc.

Dustin, E., Rashka J. & Paul J. 1999. Automated Software Testing. Reading, Massachusetts. Addison-Wesley Longman, Inc.

Endres, A. & Rombach D. 2003. A Handbook of Software and Systems Engineering. United Kingdom. Pearson Education Limited.

Fewster, M. 2001. Common Mistakes in Test Automation. Grove Consultants. United Kingdom. Viitattu 28.11.2016, http://cm.techwell.com/sites/default/files/articles/XDD2901filelistfilename1_0.pdf.

Francino, Y. 2015. The test automation basics every software developer should know. Viitattu 28.11.2016, http://techbeacon.com/test-automation-basics-every-software-developer-should-know.

Ghahrai, A. 2015. Test Automation Advantages and Disadvantages. Viitattu 28.11.2016, http://www.testingexcellence.com/test-automation-advantages-and-disadvantages/.

Ghanakota, G. 2012. Testing Frameworks. Viitattu 28.11.2016, http://www.cs.colorado.edu/~kena/classes/5828/s12/presentations/testing-frameworks-by-gayat.html.

GSMA. The Mobile Economy 2016. Viitattu 28.11.2016, http://www.gsmamobileeconomy.com/GSMA_Global_Mobile_Economy_Report_2016.pdf.

Hutcheson, M L. 2003. Software Testing Fundamentals. Wiley Publishing Inc. Indianapolis, Indiana.

Jenkins. 2016. Viitattu 28.11.2016, https://jenkins.io/.

Kaner, C. 2000. Architectures of Test Automation. Viitattu 28.11.2016, http://kaner.com/pdfs/testarch.pdf.

Katz S., Dabrowski C., Miles K. & Law M. 1994. Glossary of Software Reuse Terms. Computer Systems Technology. U.S. Department of Commerce. Viitattu 28.11.2016, http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-222.pdf.

Laukkanen, P. 2006. Data-Driven and Keyword-Driven Test Automation Frameworks. Master's thesis. Helsinki University of Technology. Viitattu 28.11.2016, http://eliga.fi/Thesis-Pekka-Laukkanen.pdf.

Loveland S., Miller G., Prewitt Jr. R. & Shannon M. 2005. Software Testing Techniques – Finding the Defects that Matter. Hingham, Massachusetts. Charles River Media, Inc.

Mazurkiewicz, M. 2010. GUI Test Automation With SWBOT. Opinnäytetyö. Vaasan ammattikorkeakoulu.

Meszaros, G. 2016. Goals of Test Automation. Viitattu 28.11.2016, http://xunitpatterns.com/Goals%20of%20Test%20Automation.html.

Nokia 2016. Liiketoimintamme. Viitattu 28.11.2016, http://company.nokia.com/fi/liiketoimintamme.

Nokia Create Process – Delivery Types and Naming Policy. 2016. Sisäinen lähde. D431005093, versio 4.1.

Nokia Create Process – General testing overview. 2013. Sisäinen lähde. D436568953, versio 2.1.

Nokia Create Process – Integration and Verification (I&V): Training, test levels and areas part. 2016. Sisäinen lähde. D540868451, versio 1.

Nokia Create Process – Introduction. 2016. Sisäinen lähde. D425832820, versio 11.1.

Nokia Create Process – SW Development. 2016. Sisäinen lähde. D425878598, versio 2.2.

Nokia Create Process – Training material: I&V Process in iterative development mode. 2013. Sisäinen lähde: D425878668, versio 2.1.

Nokia NetAct. 2016. Viitattu 28.11.2016, https://networks.nokia.com/solutions/netact.

Nokia NetAct 16.2 Single RAN Management Overview. 2016. Sisäinen lähde. DN09212604, Issue 2-0.

Nokia NetAct 16.2 System Overview. 2016. Sisäinen lähde. DN09136965, Issue 2-1.

Nokia Networks Online Services. 2016. Viitattu 28.11.2016, https://online.networks.nokia.com.

Nokia Single RAN Management Overview. 2016. Sisäinen lähde. DN09212604, Issue 2-0

Nokia Single RAN 16.2. Migration to SBTS and Commissioning. 2016. Sisäinen lähde. DN09226638, Issue 01C.

Nokia Single RAN System Description 16.2. 2016. Sisäinen lähde. DN09185655, Issue 01A.

Nokia SRAN 16.2 Master Test Plan. 2015. Sisäinen lähde. D530503370, versio 13.

Optimus Information. 2016. Traditional vs Agile Software Development. Viitattu 28.11.2016, http://www.optimusinfo.com/traditional-vs-agile-software-development/.

Piiroinen, I. 2006. Testausautomaatio Java-sovelluksen graafisen käyttöliittymän testauksessa. Diplomityö, Oulun yliopisto.

Päätalo, H. 2015. Tuoteprosessi. Master-opinnot 2015. Teknologialiiketoiminta. Oulun ammattikorkeakoulu.

Ravichandran, T. & Rothenberger M A. 2003. Software reuse strategies and component markets. Communications of the ACM. Viitattu 28.11.2016, http://dl.acm.org/citation.cfm?id=859678.

Robot Framework 2016. Viitattu 28.11.2016, http://robotframework.org/.

Sanakirja 2016. Viitattu 28.11.2016, http://www.sanakirja.org/search.php?id=81296&l2=17.

Software Testing Help. 2016. Most Popular Test Automation Frameworks with Pros and Cons of Each. Viitattu 28.11.2016, http://www.softwaretestinghelp.com/test-automation-frameworks-selenium-tutorial-20/.

Sommerville, I. 2004. Software Reuse. Viitattu 28.11.2016, https://ifs.host.cs.st-andrews.ac.uk/Books/SE7/Presentations/PDF/ch18.pdf.

Spolsky, J. 2000. Things You Should Never Do, Part I. Viitattu 28.11.2016, http://www.joelonsoftware.com/articles/fog0000000069.html.

Thomas, K. 2009. Core reuse is overrated. Viitattu 28.11.2016, http://asserttrue.blogspot.fi/2009/03/code-reuse-is-overrated.html.

Vogel, F. 2014. Why PE And Agile Haven't Always Gone Together. Viitattu 28.11.2016, https://techmvp.net/2014/07/04/agile-pe/.

Webperformance, 2016. Types of Element Locators. Viitattu 28.11.2016, http://www.webperformance.com/load-testing/blog/real-browser-manual/building-a-testcase/how-locate-element-the-page/type-element-locators/.

Yle. 2016. Verkossa on ruuhkaa. Viitattu 28.11.2016, http://yle.fi/uutiset/verkossa_on_ruuhkaa__operaattori_joudumme_tarkistamaan_suunnitelmia_joka_viikko/8985968.