OAMK
OULUN AMMATTIKORKEAKOULU

Tommi Pekkala

TESTIAUTOMAATIOYMPÄRISTÖN LUONTI QT:N KOMPONENTEILLE QNX 7.0 -REAALIAIKAKÄYTTÖJÄRJESTELMÄÄN

TESTIAUTOMAATIOYMPÄRISTÖN LUONTI QT:N KOMPONENTEILLE QNX 7.0 -REAALIAIKAKÄYTTÖJÄRJESTELMÄÄN

Tommi Pekkala
Opinnäytetyö
Kevät 2019
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, laite -ja tuotesuunnittelu

Työn tekijä: Tommi Pekkala
Opinnäytetyön nimi: Testiautomaatioympäristön luonti Qt:n komponenteille QNX 7.0 -reaaliaikakäyttöjärjestelmään.
Työn ohjaajat: Kari Hormi The Qt Company, Eino Niemi OAMK
Työn valmistumislukukausi ja -vuosi: Kevät 2019
Sivumäärä: 43

Tämä opinnäytetyö käsittelee testiautomaatiokokonaisuuden suunnittelun ja rakennuksen eri vaiheet. Tavoitteena on käydä läpi toimivan testiautomaation edellytykset ja eri tekijät, jotka voivat johtaa huonoon testiautomaation.

Tässä työssä rakennettavan testiautomaatiokokonaisuuden on tarkoitus testata Qt:n toimivuus QNX-reaaliaikakäyttöjärjestelmässä ja testata, että kokoonpanolla voidaan kääntää lähdekoodi ja lähettää se Intel NUC -tietokoneeseen. Testikokoelma myös testaa muutaman esimerkkisovelluksen toimivuuden kyseisessä ympäristössä.

Opinnäytetyö tehtiin toimeksiantona The Qt Companylle.

Asiasanat: Qt, QNX, ohjelmistotestaus, Squish

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Option of Equipment and Product Design

---

Author: Tommi Pekkala
Title of thesis: Creating test automation suite for testing Qt on QNX 7.0 real-time operating system
Supervisors: Kari Hormi The Qt Company, Eino Niemi OAMK
Term and year when thesis was submitted: Spring 2019
Number of pages: 43

This thesis explains the steps for designing and building a test suite. The goal is to bring the prerequisites into the light and explain reasons that may lead to bad test automation.

The purpose of the test suite is to verify the functionality of Qt in QNX real time operating system and test that compiling and deploying to Intel NUC target device works. The test suite also tests that applications run in the mentioned environment.

The thesis was done by request for The Qt Company.

Keywords: Qt, QNX, software testing, Squish

ALKULAUSE

Tahdon kiittää vanhempaa ohjelmistokehittäjää Sami Nurmenniemeä siitä, että hän selvensi asioita ja auttoi työn aikana erinäisissä haasteissa. Lisäksi tahdon erityisesti kiittää ohjelmistokehittäjä Kari Hormia työni valvonnasta, ohjeista, vinkeistä ja kommenteista. Kiitos kuuluu myös ohjaajalleni lehtori Eino Niemelle, joka luki ja antoi kommentteja työhöni.

Tahdon myös kiittää The Qt Companya siitä, että he ovat minut palkanneet ja näin ollen olen päätynyt tekemään mielenkiintoisia projekteja.

Oulussa 1.4.2019

Tommi Pekkala

SISÄLLYS

TIIVISTELMÄ 3
ABSTRACT 4
ALKULAUSE 5
SISÄLLYS 6
SANASTO 7
1 JOHDANTO 8
1.1 Julkaisuvaiheen testiautomaatio 8
1.2 Työn tavoite 10
2 SUUNNITTELU 12
2.1 Nykyinen arkkitehtuuri 12
2.2 Squish-testaustyökalu 15
2.3 Tuleva arkkitehtuuri 16
2.4 Arvioidut haasteet 18
2.5 Työn osa-alueet 19
3 TESTIAUTOMAATIOYMPÄRISTÖN TOTEUTUS 21
3.1 Qt:n asennus 21
3.2 QNX-levykuvan luonti 23
3.3 Kuvan kirjoitus 27
3.4 Squish-serveritiedosto 29
3.5 QNX-laitteen lisäys 29
3.6 Laitteen testaus 31
3.7 Testattavan laitteen ympäristön alustus 31
3.8 Sovellustestaus 34
3.9 Sovellustestaustyypit 36
3.10 Vianetsintätyökalun testaus 37
3.11 Testiympäristön alustus 38
3.12 Jenkins-integraatio 38
3.13 Toimivan kokonaisuuden testaus 39
4 YHTEENVETO 41
LÄHTEET 42

SANASTO

BlackBerry QNX = reaaliaikakäyttöjärjestelmä

Git = versionhallintatyökalu

GUI = graafinen käyttöliittymä

Jenkins = avoimen lähdekoodin automaatioserveri

Julkaisuarkisto = eräänlainen pilvipalvelu ohjelmakoodille (repositorio), julkaisuarkistoa käytetään yhdessä versionhallinnan kanssa

Linux = käyttöjärjestelmä

Python = tulkattava skriptikieli

Qt Creator tai Creator = Qt:n tarjoama ohjelmistokehitysympäristö

RTA = Release Test Automation, ohjelmiston viimeinen testaustaso

SD-kortti = siirrettävä muistiväline

Shell = komentotulkki

SSH = Secure Shell -yhteys

1 JOHDANTO

Tämä opinnäytetyö tutkii testiautomatiikan rakentamisen haasteita ja niiden ratkaisukeinoja. Työ tulee tuottamaan kohdeyritykselle käyttöön valmiin testiautomaatiokokonaisuuden. Kohdeyritys on ohjelmistoja valmistava monikansallinen yritys The Qt Company, jonka päätuote on Qt.

Qt on oliopohjainen C++-ohjelmointikielellä kirjoitettu järjestelmäriippumaton ohjelmistokehys, jolla voidaan kehittää ohjelmistoja (About Qt. 2018). Ohjelmistokehys helpottaa ohjelmistokehittäjän työtä tarjoamalla valmiita työkaluja. Ohjelmistokehys tarjoaa mm. valmiita projektipohjia graafisten sovellusten tuottamiseen.

Yritys kehittää ohjelmistokehyksen ympärille tukea eri liiketoimialueiden tarpeisiin. Qt:n teknologioilla voidaan kehittää sovelluksia ja laitteita yhden koodipohjan avulla kaikilla järjestelmillä (iOS, Windows, Linux). Autojen viihdejärjestelmät, puettava elektroniikka ja IoT-laitteet ovat vain esimerkki kaikista niistä järjestelmistä, joita voidaan kehittää. (About Us. 2018.)

Tämä mittava ohjelmistotekniikan määrä asettaa laadunvarmistuksen kannalta haasteita. Ei-toivottuja asioita laadun kannalta ovat esimerkiksi uuden ohjelmistojulkaisun mukana tullut ohjelmiston hidastuminen tai pahimmassa tapauksessa ohjelmiston väliaikainen täydellinen rikkoutuminen, joka voi johtua vain pienestä kirjoitusvirheestä tai ohjelmistopolun muuttumisesta.

1.1 Julkaisuvaiheen testiautomaatio

Testiautomaatio on automatisoitua testausta. Testiautomaatio suoritetaan käyttämällä jotain erityistä ohjelmistoa toisen ohjelmiston ohjailuun, jotta voimme luoda vertailtavia tapahtumia (Test Automation. 2018). Testiautomaatiolla voidaan automatisoida toistuvien, mutta tärkeiden toiminnallisuuksien toisto ja oikeellisuuden varmistus. Testiautomaatio voi olla erittäin hyödyllistä oikein tehtynä. Se voi olla myös täysin hyödytöntä ja väärän turvallisuudentunteen antavaa, mikäli sitä ei ole tehty oikein. Mikäli testiautomaatio antaa virheellisiä hälytyksiä

tai ei anna hälytyksiä, on testiautomaation hyöty kyseenalainen. Tämänkaltaisessa tapauksessa testiautomaatio saattaa jäädä käyttämättä. (Pettichord 1996, 7.)

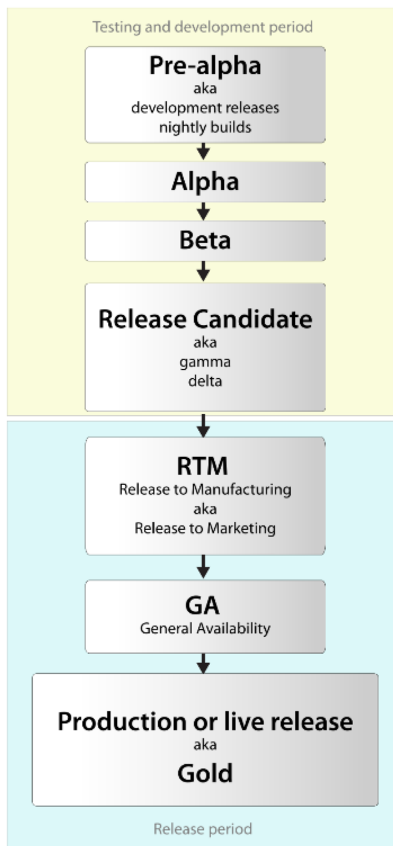
Testiautomaatiolla voidaan haluta testata koko tuote tai osa tuotteen ominaisuuksista. Testattavassa ohjelmassa saattaa olla useita eri rajapintoja, joita halutaan testata. Erilaisia rajapintoja ovat mm. graafinen käyttöliittymä, ohjelmointirajapinta tai komentorivi (Pettichord 2001, 7). Testiautomaation yleinen pilaava tekijä on testaajan kokemuksen puute. Tällöin kyseessä saattaa olla esimerkiksi kokematon testaaja, jonka mielenkiintona on ohjelmointi testaamisen sijasta. Tämänkaltaisessa tilanteessa saattaa työstä kadota fokus. (Pettichord 1996, 2.)

Pilaavia tekijöitä saattaa olla myös testiautomaation rakentaminen muun työn sivussa. Tämä estää työtä saamasta sitä huomiota, minkä se tarvitsisi. Myös tarkkojen tavoitteiden ja päämäärien puuttuminen katsotaan huonoksi (Pettichord 2001, 2). Testiautomaatio tulee rakentaa helposti ylläpidettäväksi ja ymmärrettäväksi. Testiautomaation tulisi voida olla sellaista, että sen voi luovuttaa helposti toiselle testaajalle käyttöön.

Testiautomaatio toimii ideaalisesti siten, että automatiikka saa ilmoituksen uuden version saapumisesta julkaisuarkistoon, jonka jälkeen ohjelmisto ottaa haltuun version latauksen ja kaikki sen alustukseen sekä testaukseen liittyvät seikat, mukaan lukien raportoinnin.

Oikein tehdyllä automatisoinnilla voidaan vähentää puuduttavien työtehtävien määrää henkilöstöltä ja saada luotettavia tuloksia. Testaamisen kehittäminen automaattiseksi poistaa ihmisvirheen testauksesta sekä mahdollistaa yhtenäisten testitulosten kasaamisen arviointia varten. Näin ollen voidaan luoda esimerkiksi datasetti ohjelmiston suorituskyvystä eri versionumeroiden välillä.

Julkaisuvaiheen testiautomaation tärkein tehtävä on varmistaa julkaisukandidaatin julkaisukelpoisuus (kuva 1). Julkaisuvaiheen testaus on testauksen viimeisin vaihe tuotteelle ennen kuluttajaa. Mikäli testaus paljastaa tuotteessa vakavan puutteen, on tuotteessa julkaisueste. Jos tuote menee onnistuneesti läpi, tarkoittaa tämä, että tuotteen voi julkaista.



KUVA 1. Sovelluskehityksen vaiheet (Software release life cycle. 2019)

Julkaisuvaiheen testiautomaatio eli Release Test Automation (RTA) on tärkeytensä vuoksi osa yrityksen laadunvarmistuksen prosesseja. Julkaisuvaiheen testiautomaatiolla pyritään havaitsemaan laatua heikentävät tekijät ennen lopullista julkaisua.

1.2 Työn tavoite

Testiautomaatiokokonaisuudella tarkoitetaan useista eri skripteistä koostuvaa ohjelmakoodikokonaisuutta, jolla on monia eri toimintoja, mutta yksi tarkoitus. Tarkoitus tässä työssä on luoda sulava testiskriptikokoelma, joka testaa kaikkien Qt:n tarjoamien oleellisten työkalujen ja kirjastojen toiminnan BlackBerry QNX -reaaliaikakäyttöjärjestelmässä. Työhön kuuluvat seuraavat osa-alueet: tiedostojen lataus ja ympäristön alustus, Qt Creatorin asennus, Qt-kehityskirjastojen asennus, levykuvan kirjoitus, sovellusten käyttöönotto Intel NUC -tietokoneeseen

ja muutamien sovellusten testaus. Nämä osa-alueet tullaan tekemään automaattisesti. Osa-alueiden testaus varmistaa, että kaikki peruskäyttöön liittyvät toiminnot toimivat. Työtä tarvitaan, koska yhtiö tukee QNX-käyttöjärjestelmää.

Kohdeyrityksessä on käytössä testiautomaatiota entisestään, joten työssä pidetään lähtökohtana sitä, että uuden pitää olla integroitu aikaisemmin rakennettuihin rakenteisiin tai sen pitää muistuttaa aikaisemmin rakennettua, jotta ylläpito on helppoa. Testaus tulee tapahtumaan Ubuntu 16.04 -käyttöjärjestelmällä. Kokemusta kuitenkin on, että ohjelmistoa voidaan tarvita muissakin ympäristöissä, mikäli halutaan käyttää isäntänä vaikkapa Windows-käyttöjärjestelmää. Tällöin opinnäytetyössä käytetään skriptauskielenä mahdollisimman paljon Pythonia, sillä sitä voidaan ajaa useissa eri ympäristöissä. Työssä käytetään myös Shell-skriptejä, joilla voidaan ohjata käyttöjärjestelmän toimintoja.

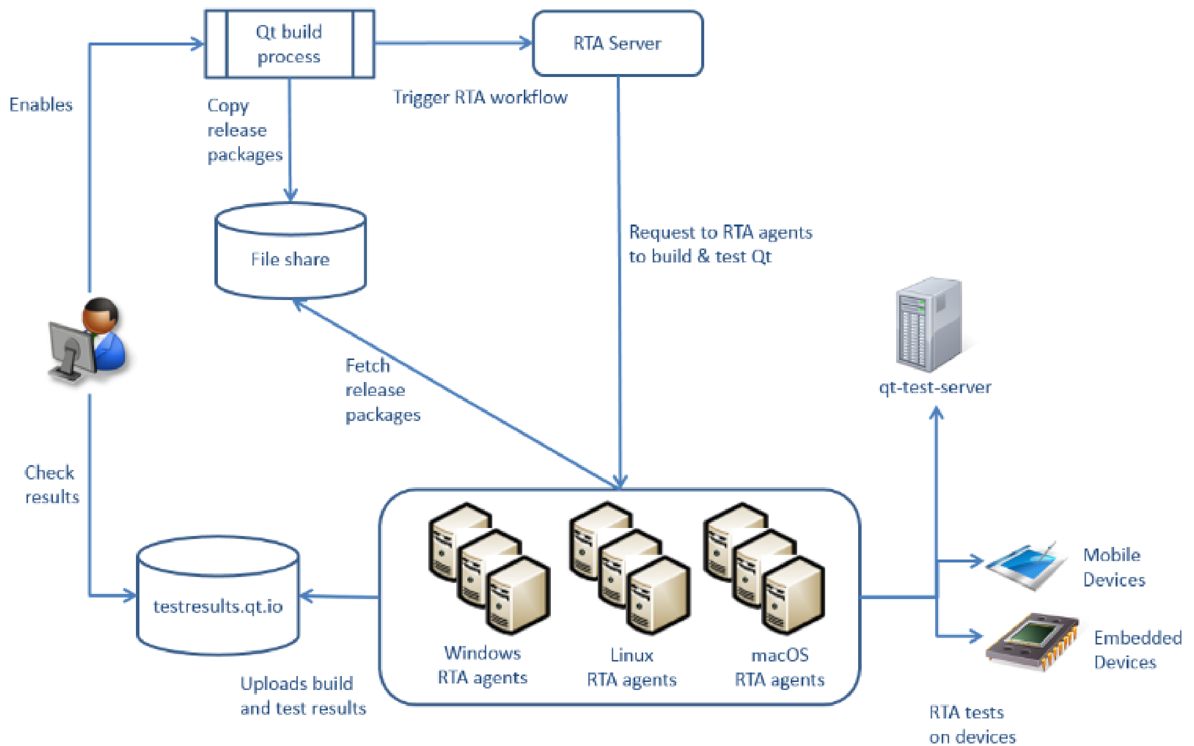
2 SUUNNITTELU

2.1 Nykyinen arkkitehtuuri

Yrityksessä on käytössä entisestään testiautomaatiota. Testiautomaatio suoritetaan pääosin Squish-testaustyökalulla, joka on testaukseen erikoistunut ohjelmistokehitys-testausympäristö. Ohjelmistotestauskoodi on testaukselle allokoidussa julkaisuarkistossa, jossa se on edelleen jaettu kansioihin eri käyttökohteiden mukaisesti. Julkaisuarkistossa on myös muita testaukseen liittyviä ohjelmistokokonaisuuksia ja skriptejä. Tämän takia työssä käytetään myös jo olemassa olevia funktiota, joita sitten muokataan tarpeen mukana.

Nykyisessä arkkitehtuurissa (kuva 2) kaikki testaustyöt ovat keskitetty Jenkinsiin. Jenkins on avoimen lähdekoodin integraatiotestausalusta, jolla automatisoidaan ja organisoidaan töitä. Jenkins työ voidaan käynnistää manuaalisesti tai automaattisesti, kuten luvussa 1.1 mainittiin. Testausprosessien hallinnointi kuuluu yleensä henkilölle, joka kyseiset testit on kirjoittanut.

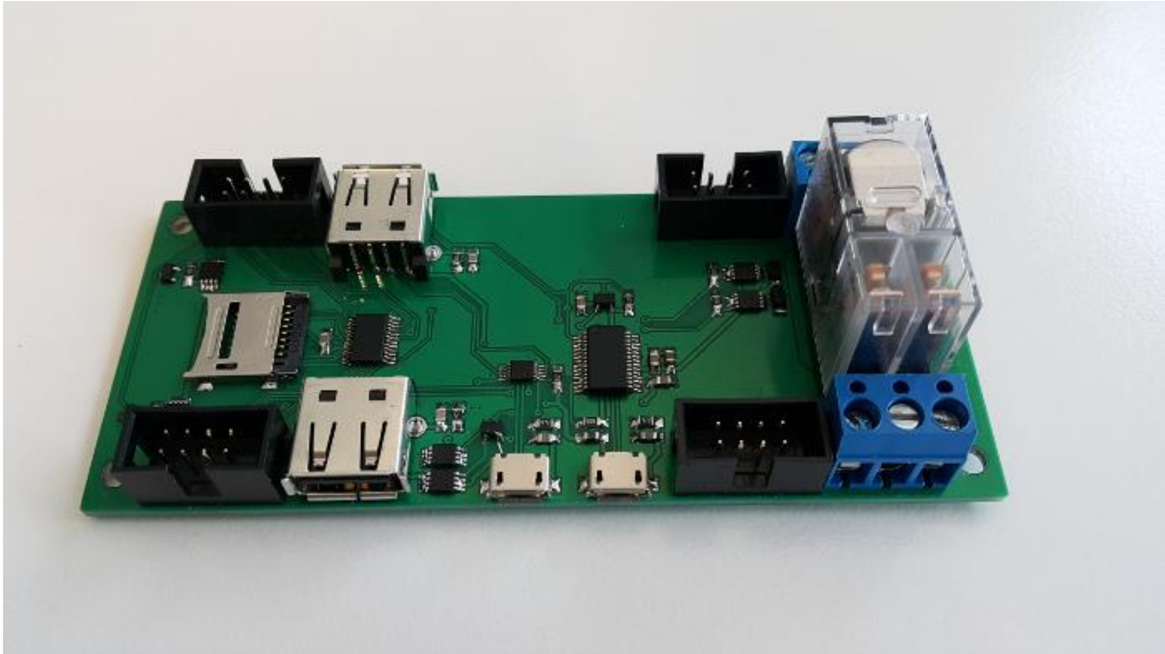
Jenkinsillä on myös melko laaja yhteisön tuki ja Jenkinsiä käytetään ohjelmistoalalla yleisesti. Alusta toimii siten, että Jenkins-isäntäohjelmisto eli master jakaa agenteille tehtävät, jotka sitten työstävät tulokset isännälle. Jenkinsiä voi myös käyttää pelkästään isäntätilassa, jolloin se tekee kaiken työn. Tämä käy yksinkertaisiin ja pieniin konfiguraatioihin. Agentteja käytetään, mikäli isäntä haluaa siirtää itseltään pois töitä ja vapauttaa omia laitteistoresurssejaan. Agenteilla tarkoitetaan tässä kontekstissa fyysistä tai virtuaalista tietokonetta, jossa pyörii Jenkins -agentti taustalla kuuntelemassa isännän käskyjä. (Kawaguchi 2018.) Tämä työ tullaan liittämään Jenkins-alustaan, josta testaaja laukaisee testikokoelman käyntiin. Prosessi alkaa uusimmat testausskriptit sisältävän julkaisuarkiston kloonaamisesta agenttikoneelle, jonka jälkeen käynnistyy Qt-asennuspaketin lataus ja sen jälkeen itse testikokoelma. Kloonattava julkaisuarkisto sisältää kaiken testaamisen tarvittavan ohjelmakoodin.



KUVA 2. Testiautomaatioprosessi (Qt Quality Management 2018)

Testauksen ollessa täysin automatisoitua tulee myös testattavan laitteen toimintojen olla automaattisia fyysisessä mielessä. Tällä tarkoitetaan SD-kortin siirtelyä sekä virtojen kytkemistä päälle ja pois. Virran tulee olla pois päältä, kun uusi levykuva kirjoitetaan SD-kortille, ja se laitetaan päälle, kun levykuva on kirjoitettu ja laitteessa kiinni.

Tähän ongelmaan on olemassa ratkaisu, joka on jo käytössä. Ratkaisuna on käytetty SD-MUX-piiriä (kuva 3), jonka nimi tulee sanoista Secure Digital Multiplexer (SD-MUX 2018). Piiri säätelee laitteen virtaa sekä SD-kortin sijaintia.



KUVA 3. SD-kortinvaihtaja (SD MUX 2018)

Piiriä käytetään sd-mux-control -nimisellä komentoriviohjelmalla, joka siirtää SD-kortin paikan isäntäkoneesta testikoneeseen ilman manuaalista toimenpidettä. Piiri kytketään isäntäkoneeseen kahdella mikro-USB -liittimellä. Virtojen ohjaus tapahtuu kuvassa 3 oikealla olevan releen kautta. SD-kortin siirtely sekä virtojen hallinta voidaan suorittaa komentorivipohjaisesti, mikä mahdollistaa automatisoinnin.

Testikokoelma koostuu useista eri testeistä, jotka voivat toimittaa useita testaamisen liittyviä tarpeita. Testikokoelma on ohjelmistokokonaisuus, joka on rakennettu jonkin tietyn aihealueen testaamiseen. Testikokoelma voi olla vaikkapa kokoelma tarvittavia toimintoja, jotta voidaan varmistaa, että asennusohjelma toimii oikein. Testit eivät yleensä suorita pelkästään vertailuja, vaan ne myös tekevät toimintoja. Toimintoja voi olla vaikkapa muuttujien lisääminen tiedostoon tai käyttöjärjestelmän ohjaus.

Testausarkkitehtuurissa on yleensä ollut käytössä jokin pääkirjaston virkaa tekevä tiedosto, johon on kerätty oleellisimmat testaukseen liittyvät funktiot. Oleellisia funktioita ovat muun muassa Creatorin käyttöön liittyvät toiminnot, joita voi olla vaikkapa kääntäjän ulostulosyötteen talteenotto tai sovelluksen käynnistys.

2.2 Squish-testaustyökalu

Squish on GUI-testaustyökalu, joka on erikoistunut käyttöliittymien automaattiseen testaamiseen (Company, About froglogic.). Squish-työkalua voi käyttää useiden eri ohjelmointikielten kanssa: Python, JavaScript, Ruby, Perl tai TCL (Multiple Real-World Scripting Languages).

Qt-sovellusten objektit tallentuvat Squish-kehitysympäristön oleelliseen tiedostoon nimeltä names.py. Tähän tiedostoon tallentuvat Squish-työkalun käsittelemät objektit. Object map eli objektikartta on erittäin käytännöllinen ohjelmistotestauksessa, sillä se sallii muuttuvan eli dynaamisen käyttöliittymän testauksen. Vaikka painikkeiden ja tekstien sijainnit vaihtuisivatkin käyttöliittymässä, Squish löytää ne objektikartan avulla. Objektikartat esitetty kuvissa 4 ja 6. Squish-sovellusversio päivittyy 6.4-versioon tämän työn kirjoituksen aikaan ja tuo samalla päivityksen objektikartan käyttöön. Objektikartta muuttuu tekstipohjaisesta skriptipohjaiseksi ja näin ollen muuttaa objekteihin viittaamisen. Tämä ei ole työn kannalta ongelma.

```
options_Core_Internal_SettingsDialog = {"type": "Core::Internal::SettingsDialog",\
 "unnamed": 1, "visible": 1, "windowTitle": "Options"}
```

KUVA 4. Skriptipohjainen objektikartta

```
names.options_Core_Internal_SettingsDialog
```

KUVA 5. Viittaus skriptipohjaiseen karttaan

```
:Billboard_Text {container=':list_Item_2' text='Billboard'\
 type='Text' unnamed='1' visible='true'}
```

KUVA 6. Tekstipohjainen objektikartta

```
:Billboard_Text
```

KUVA 7. Viittaus tekstipohjaiseen karttaan

2.3 Tuleva arkkitehtuuri

Tuleva kokonaisuus koostuu seuraavista pääosista: pakettien lataus, niiden asennus, ympäristön asetus ja testaus. Työ muistuttaa rakenteeltaan muita yrityksen testikokonaisuuksia, sillä peruspiirteet ovat samat. Vaikka Squish tarjoaa mahdollisuuden testitapausten luomiseen nauhoittamalla, tähän ei tulla täysin luottamaan. Testitapausten nauhoittamisella tarkoitetaan sitä, että luodaan uusi testitapaus ja aletaan nauhoittamaan ruudulla tapahtuvia toimenpiteitä. Tämä tapa ei ole kovin toimiva ylläpidettävyyden kannalta (Pettichord 1996, 2). Mikäli testit ovat esimerkiksi nauhoitettu 5.11.1-versiolla, jää testiskriptiin kovakoodatut versionumerot. Uuden ohjelmistopäivityksen myötä versionumero muuttuu ja testi ei enää löydä 5.11.1-nimisiä objekteja ja näin ollen testiautomaatio menee rikki. On järkevämpää nauhoittaa osa testeistä ja tehdä niihin dynaamisia osia esimerkiksi versionumerointiin

Ajatus on, että testikokonaisuus menee /suites/qnx/qnx_testsuite-polkuun ja /suites/qnx/qnx_testsuite/shared/defines.py-polkuun tulee yleisfunktiokirjasto, joka tarjoaa kaikki pääfunktiot. Sisällytetään linkit olemassa oleviin asentajaskripteihin, jotta saadaan niiden funktiot hyötykäyttöön Defines.py-tiedostoon.

Tarvittavat Shell-skriptit tulevat sitten tämän kokonaisuuden kylkeen. Näitä skriptejä säilytetään /qnx/shared-polussa. Alkutietojen perusteella testausympäristö rakennetaan yhden testattavan laitteen ja yhden käyttöjärjestelmän ympärille, mikä vähentää ehtolauseiden ja ohjausrakenteiden määrää.

Testausympäristön käytössä tulee olemaan ongelmatilanteita, ja onkin tärkeää saada ne analysoitua nopeasti. Python-ohjelmointikielessä on olemassa ainakin kahdenlaisia virheitä: syntaksivirheitä ja poikkeuksia. Syntaksivirhe on ns. parsimisvirhe, mikä kertoo koodin olevan virheellistä. Näitä virheitä kokenut koodaaja tekee vähän ja ne tulevat ilmi nopeasti. Poikkeusvirheet ovat virheitä, jotka syntyvät ajon aikana virheellisistä syötteistä tai mahdottomista komennoista. Esimerkkinä nollalla jakaminen on ZeroDivisionError. Koodi voi olla siis oikein kirjoitettua, mutta silti tuottaa virheen. (Errors and exceptions. 2018).

Poikkeuksien kaappaamista varten käytetään Pythonin tarjoamaa yritä-poikkeus -syntaksirakennetta virheenkaappausta varten. Mikäli havaitaan, että jokin kohta

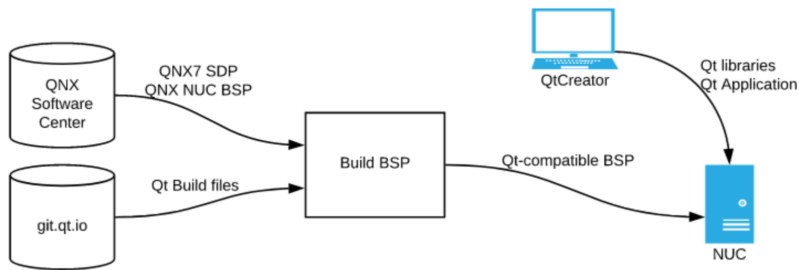
ohjelman suorittamisessa voi olla epävakaa, laitetaan kuvassa 8 esitelty yritä -poikkeus rakenne ohjelmakoodiin. Sen avulla voidaan vähentää ohjelman kaatumisesta aiheutuvia ongelmia ja saada selville ajossa tapahtuneiden virheiden perimmäiset syyt ja sijainnit. Ohjelmarakenne vastaa C++-ohjelmoijille tuttua try-catch-rakennetta. Virheiden hallinnoiminen nopeuttaa kehitystyötä ja on käytännöllistä lopputuotteessa. Ajon aikaisten virheiden hallinnointi mahdollistaa virheiden paikannuksen ja näin ollen voidaan päätellä, johtuuko virhe testikoodista vai onko tuotteessa oikea virhe. Esimerkiksi tiedostojen lataus tullaan toteuttamaan kyseisellä rakenteella.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

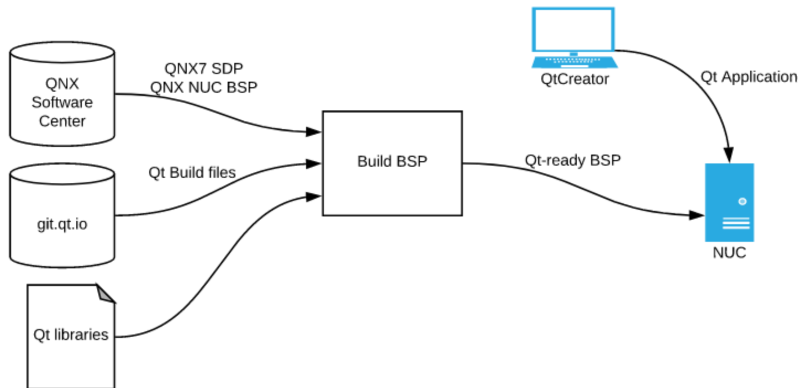
KUVA 8. Try Except -syntaksia (8. Errors and Exceptions 2018)

Työn suunnitteluvaiheessa pitää miettiä, millä tavoin Qt:n tarjoamat kirjastot tullaan sisällyttämään käyttöjärjestelmään. Päätapoja on kaksi: kirjastot voidaan kirjoittaa käyttöjärjestelmän levykuvaan, tai ne voidaan kopioida laitteelle SSH-yhteyden kautta (kuva 9).

Method A: Deploy Qt libraries with QtCreator



Method B: Include Qt libraries in BSP



KUVA 9. Qt kirjastojen sisällytys (QNX 7.0 on Intel NUC. 2018)

Levykuvaan sisällyttämisessä ei ole haittapuolia, kun taas verkon yli tiedostojen siirto voi joskus epäonnistua. Testikokoelmasta halutaan mahdollisimman vakaa, joten levykuvaan sisällyttäminen on luonnollinen vaihtoehto.

2.4 Arvioidut haasteet

Työ tulee olemaan rakenteeltaan osittain erilainen kuin aikaisemmin rakennetut testausympäristöt. QNX-käyttöjärjestelmä tulee vaatimaan tavanomaista enemmän erilaisten ympäristömuuttujien määrittämistä ja laitteen automaattista ohjaamista SSH-yhteyden kautta. Laite vaatii salasanan syöttämisen SSH-yhteyden muodostamisen yhteydessä, joten siinä on varmasti mielenkiintoinen haaste.

Aikaisempaa kokemusta Squish-työkalun käyttämisestä reaaliaikakäyttöjärjestelmässä ei ole, joten siinä riittää varmasti opiskeltavaa ja tutkittavaa. Pyrin työssä myös vakauteen ja hyvään tulostenantokykyyn. Testikokoelma tehdään selaiseksi, että siitä on oikeasti hyötyä. Olen havainnut, että Squish-työkalulla on

vaikeuksia löytää objekti, mikäli jokin parametri objektissa muuttuu. Tämä pyritään korjaamaan tässä työssä.

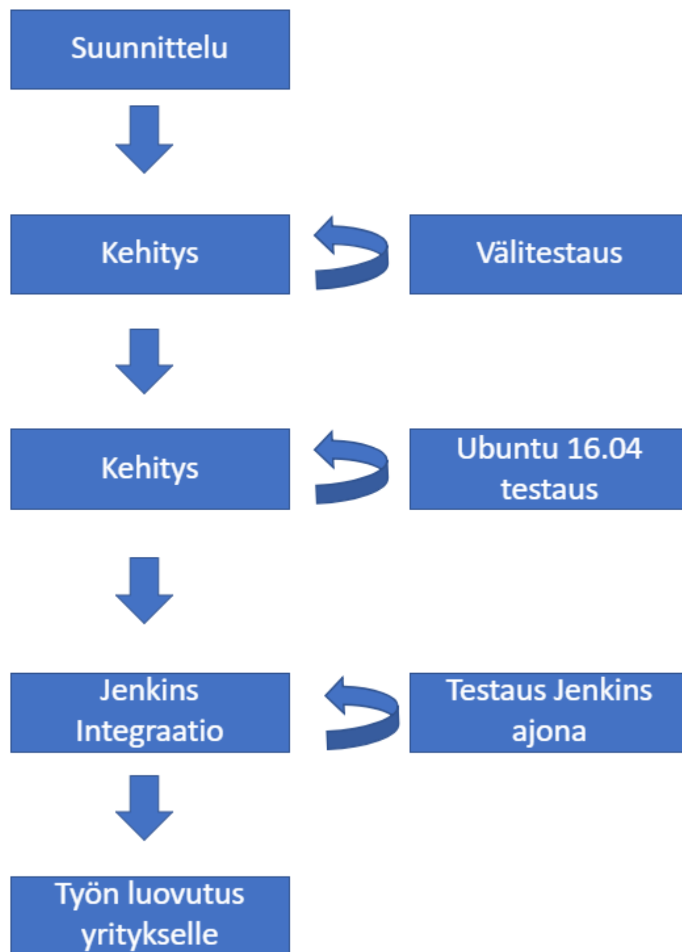
2.5 Työn osa-alueet

Työ koostuu eri osa-alueista (kuva 10), joilla varmistetaan työn valmistuminen ajoissa ja lopputuloksena hyvä laatu.

Työssä on kolme testausympäristöä, joilla voidaan testata testikokonaisuutta sen eri vaiheissa. Kevyitä väliaikaistestejä voidaan suorittaa kehityksessä käytettävällä tietokoneella; näillä testeillä saadaan pienellä vaivalla selville testitapausten toimivuus. Väliaikaistestejä ajetaan useita päivässä, jotta voidaan varmistaa testauskoodin toimivuus.

Virallisella testikoneella testaus tarkoittaa testien ajamista siinä ympäristössä ja sillä koneella, millä testit tullaan ajamaan. Nämä testiajot antavat miltei täydellisen kuvan siitä, toimivatko skriptit kyseisessä ympäristössä.

Virallisella testikoneella Jenkins-alustan kautta testaus tarkoittaa sitä, että testit ajetaan edellä mainitulla koneella, mutta Jenkinsin ajamana. Näillä ajoilla saadaan selville todellisen testiympäristön toimivuus.



KUVA 10. Työn vaiheet

3 TESTIAUTOMAATIOYMPÄRISTÖN TOTEUTUS

3.1 Qt:n asennus

Ensimmäinen testitapaus tässä testikokoelmassa tulee olemaan ladatun asennuspaketin asennus. Kaikkiin testikokoelmiin tulee Jenkins-ajon mukana asennuspaketin lataus, joten testikokoelma ei lähde liikkeelle siitä. Testikokoelman asennusympäristön perustiedot tulevat env_variables.py-nimisestä tiedostosta (kuva 11), joka kuuluu yrityksen alkuperäiseen testausarkkitehtuuriin.

Asennuksen ja asentajapaketin latauksen kannalta tärkeimmät tiedot tulevat kuvassa 11 mainituista kohdista.

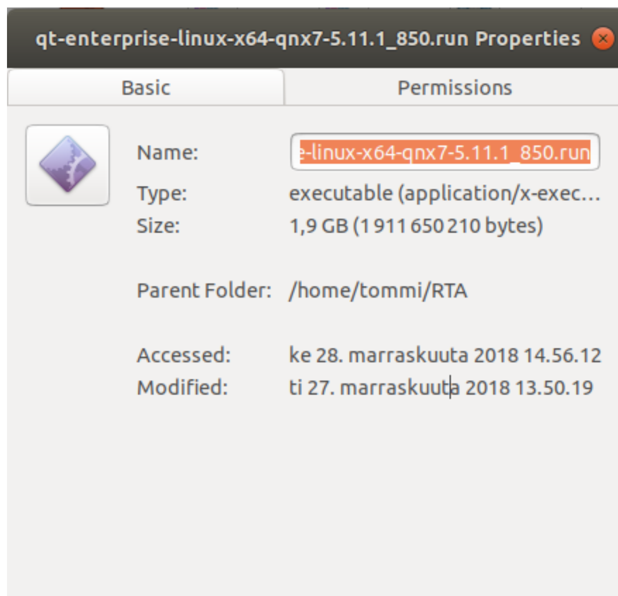
```
def get_job_name():
    return os.environ.get(envVariable_jobname, "ERROR_no_job_name_defined")

# Function returns the Qt version to be tested
# Note, if test is run locally, hard code correct Qt version here
def get_qt_version():
    return os.environ.get("QT", "ERROR_Qt_version_undefined")


# Function returns used license type, enterprise/opensource. Defaults to enterprise
# Note, if test is run locally, hard code correct license type here
def get_license():
    return os.environ.get("LICENSE", "enterprise")


# Function returns used package type, online/offline/monolite. Defaults to online
# Note, if test is run locally, hard code correct package type here
def get_installer():
    return os.environ.get("INSTALLER", "online")
```

KUVA 11. Ympäristömuuttujat

Funktio get_job_name() palauttaa testauksen työnimen. Työnimessä kerrotaan, mikä Qt-moduuli asennetaan, mitä kääntäjiä asennetaan ja onko testattava kone fyysinen vai virtualisoitu. Funktio get_qt_version() kertoo testausympäriställe testattavan Qt-version. Funktio get_installer() tietää kertoa, käytetäänkö offline vai online asentajaa.

Työssä käytettiin työnimeä: RTA_QNX700_HW_nuc/cfg=linux-g++-Ubuntu16.04-x64_QNX700-x64-PM. Työnimi koostuu kahdesta osasta, nimestä ja konfiguraatio-osasta.

Työnimen tietojen perusteella osattiin valita oikeat komponentit asennusohjelmassa, sekä ladata oikea asentajaohjelma. QNX on tällä hetkellä ainoastaan offline-asennusohjelmassa. Offline-asentajaohjelmistossa ovat kaikki tiedostot mukana, eikä asennuksen aikana tarvitse ladata mitään. Tämä johti siihen, että asennusohjelman lopulliseksi kooksi tuli kuvan 12 mukaisesti 1,9 gigatavua.



KUVA 12. Asennustiedoston koko

Testikokoelman asennusvaiheessa ohjelmisto laitetaan tarkastamaan mahdollisen aikaisemman Qt:n asennuksen kansiosta /qt/RTA ja mikäli sellainen löytyy, se poistetaan.

Testitapauksen tehtävänä on myös ympäristömuuttujien perusteella luoda asennusskripti asentajaohjelmalle, jotta asentajasta voidaan valita oikeat komponentit asennusta varten. Asentajaohjelmisto käynnistetään siis parametrina syötetyn skriptin avulla, jossa on kaikki tarvittavat tiedot asennuksesta. Tämän testitapauksen jälkeen RTA-kansiossa pitäisi olla qt5-kansio, johon Qt on komponentteineen asennettu.

3.2 QNX-levykuvan luonti

Ohjelmiston testaamista varten pitää luoda käyttöjärjestelmän sisältävä levykuva. Levykuva on .img-päätteinen binääritiedosto. Levykuvan voi kirjoittaa siirrettävälle muistille, esim. USB-tikulle tai SD-kortille, joka tietokoneeseen kytkettynä käynnistää käyttöjärjestelmän. Jotta käyttöjärjestelmä voidaan käynnistää SD -kortilta, tulee UEFI BIOS -valikosta laittaa Boot Priority -asetukseen ensimmäiseksi siirreltävä muistiväline. Käyttämämme levykuva ei tue nykyaikaista EFI-käynnistystapaa, joten samasta BIOS-valikosta joudutaan asettamaan myös Legacy Boot -asetus päälle.

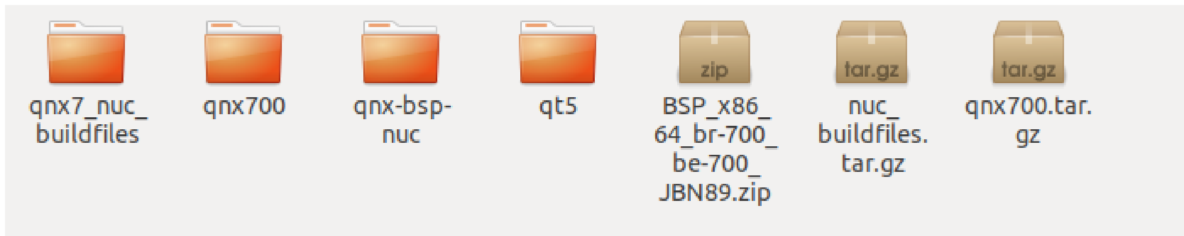
Järkevintä on luoda testitapaus, johon tulee kaikki levykuvan luontiin liittyvät operaatiot yhteen skriptiin. Tämä helpottaa kokonaisuuden hahmottamista ja vikojen löytämistä. Testitapauksen sisällä osa-alueet voi jakaa omiin funktioihinsa.

Testitapauksen pääfunktio main() tulee alkamaan tiedostojen latauksesta. Kuvan luontia varten tarvitaan erinäisiä tiedostoja, jotka pitää ladata palvelimelta. Tätä varten on Pythonissa urllib-kirjasto URL osoitteiden avaukseen ja lataukseen. Kirjaston funktiota on hyvä käyttää yritä-poikkeus rakenteessa (kuva 13), käyttäen poikkeus-kohdassa test.warning() -funktiota, jolla saadaan näkyvä virheilmoitus.

```
qnx700 = "http://ci-files01-hki.intra.qt.io/input/qnx/qnx700.tar.xz"

fileopener = urllib.URLopener()
try:
    fileopener.retrieve(qnx700, "qnx700.tar.gz")
except:
    test.warning("could not load qnx700")
```

KUVA 13. yritä, nappaa virhe -rakenne itse käytössä

Isäntäkoneeseen pitää ladata kuvassa 14 näkyvät kansiot ja tiedostot:



KUVA 14. Ladatut kansiot

Kuvassa 14 on useita samannimisiä tiedostoja, sillä kuvassa on mukana vielä tiedostojen pakatut versiot. Testikansioon ladataan QNX-ohjelmistonkehityspaketti qnx700, levykuvan kasaustiedostot (qnx7_nuc_buildfiles), Qt:lle muokattu NUC-piiritukipaketti (qnx-bsp-nuc), sekä QNX-ohjelmistokeskuksen tarjoama yleinen piiritukipaketti 64-bittisenä x86-arkkitehtuurin: BSP_x86_64_br-700_be-700_JBN89.zip.

QNX:n tarjoama geneerinen piiritukipaketti ei toiminut suoraan, vaan sitä varten on täytynyt luoda skriptimuutoksia, jotta voidaan tukea grafiikkaa ja Qt:n kirjastoja. Tämän syyn takia tiedostoja on useita. Piiritukipaketin muutos ei kuulunut opinnäytetyön piiriin vaan se oltiin tehty jo aikaisemmin.

QNX tarjoaa valmiita yleisiä piiritukipaketteja oman ohjelmistokeskuksensa kautta, mutta ne vaativat muokkauksia toimiakseen työssä käytettävän Intel NUC-tietokoneen (kuva 15) kanssa.

KUVA 15.. Intel NUC-tietokone (Intel NUC)

Levykuvan luonti tehdään build.sh nimisellä Shell-skriptillä, joka kokoaa ladatuista tiedostoista levykuvan. Mikäli kirjastot sisällytetään levykuvaan, tulee samassa Shell-istunnossa olla ajettuna qnx700-kansiossa oleva qnxsdp-env.sh Shell-skripti, joka tuo istuntoon ympäristömuuttujia. Shell-istunnon ympäristömuuttujana tulee lisäksi olla Qt:n asennuksen mukana tulleiden kirjastojen sijainti tietokoneessa.

Tämä konfiguraatio toteutetaan siten, että qnx-bsp-nuc -tiedosto ladataan testitapauksen alussa normaalisti, mutta build.sh-skripti vaihdetaan julkaisuarkiston QNX kansiossa sijaitsevaan muokattuun build.sh-skriptiin, jossa on tiedoston alussa kuvan 16 koodirivit.

```
31 QNX700_QNXSDP=$2
32 QNX_INSTALLDIR=$3
33
34 source $QNX700_QNXSDP
35 export QT_QNX_INSTALL_DIR=$QNX_INSTALLDIR
```

KUVA 16. Build.sh skriptin muutos

Koodirivit mahdollistavat sen, että skriptille voidaan antaa parametreinä kaksi tiedostopolkua, joiden avulla voidaan luoda ympäristön alustus skriptin ajoa varten.

Näiden muutosten lisäksi x86_64.build nimisen tiedoston loppuosassa olevat kolme koodiriviä (kuva 17) pitää ottaa pois #-merkin takaa ja aktivoida. Ruutu eli #-merkki tarkoittaa skriptille koodin kommentoimista eli koodiriviä ei oteta huomioon ajossa. Mikäli merkin ottaa pois, rivi ajetaan.

```
161 ###########################################################################
162 ## Uncomment to preinstall Qt. Set QT_QNX_INSTALL_DIR pointing to the
163 ## directory qnx7_x86_64 (something like ~/Qt/5.11.1/qnx7_x86_64)
164 ###########################################################################
165 #/qt/lib=${QT_QNX_INSTALL_DIR}/lib
166 #/qt/plugins=${QT_QNX_INSTALL_DIR}/plugins
167 #/qt/qml=${QT_QNX_INSTALL_DIR}/qml
168
```

KUVA 17. Kommentoitua koodia

Muutosta varten pitää toteuttaa pieni algoritmi, joka lukee ensin tiedoston tekstirivit, jonka jälkeen se kirjoittaa uudelleen rivit, joissa ei lue yhteistä tekijää "QT_QNX_INSTALL_DIR." Tämän jälkeen tiedosto avataan uudelleen ja loppuun lisätään kuvan 17 rivit ilman #-merkkiä.

QNX-levykuvassa on normaalisti SSH-kirjautumisen yhteydessä salasanakysely. Salasanakysely on toki hyvä asia, mutta automatisoinnin kannalta hankaloittava tekijä, sillä laite kysyy salasanan aina, kun sinne siirretään scp-ohjelman avulla tiedostoja tai otetaan SSH-yhteys.

Työn alkuaikoina työssä päätettiin käyttää expect-ohjelmaa, jolla voidaan toteuttaa salasanan syöttö. Expect-ohjelmaan määritetään mitä tietoa odottaa toisesta ohjelmasta ja mitä vastata (Expect(1) - Linux man page). Kysymyksiä kuitenkin herätti se, miten Expect toimisi Windows-ympäristössä, mikäli ilmenee tarvetta testata myös Windowsilla.

Ongelma päätettiin ratkaista lisäämällä levykuvan luomisskriptiin koodirivi, joka sisällyttää isäntäkoneen julkisen SSH-avaimen luotettujen avainten listaan testikoneessa. Tässä ratkaisussa on kätevää se, että salasanakyselyä ei enää tarvitse tehdä.

Jotta työ olisi mahdollisimman organisoitu, levykuvan luonti laitetaan omaksi funktiokseen (kuva 18).

```python
def build_img():
    qnxsdp_location = workDir + "qnx700/qnxsdp-env.sh"
    qt_qnx_install_dir = workDir +"qt5" + "/" + qt_version + "/" + "qnx7_x86_64"

    os.chdir(workDir + "qnx-bsp-nuc/")
    zip_file_location = workDir + "BSP_x86_64_br-700_be-700_JBN89.zip"
    run_cmd = "./build.sh" + " " + zip_file_location + " " + qnxsdp_location + " " + qt_qnx_install_dir
    script_process = subprocess.Popen([run_cmd], shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    out = ""
    err = ""
    while script_process.poll() is None:
        comm_out, comm_err = script_process.communicate()
        out += comm_out
        err += comm_err
        snooze(1)
    out_list = out.split('\n')
    err_list = err.split('\n')
    test.log("stdout:")
    test.log("stdout and stderr currently commented out")
    for line in out_list:
        pass
        #FOR DEBUG PURPOSES
        #test.log(line)
    test.log("stderr:")
    for line in err_list:
        if "fail" in line or "err" in line or "ERR" in line or "FAIL" in line:
            test.fatal(line)
        else:
            test.log(line)
```

KUVA 18. Levykuvan luomisfunktio

Levykuvan luova Shell-komento suoritetaan käyttäen Subprocess-luokan Popen-funktiota, jolle annetaan parametrina komento. Tämä funktio on kätevä, koska se palauttaa stdout- ja stderr-syötteet. Nämä syötteet kertovat, miten prosessi on edennyt tai miksi se on epäonnistunut.

Testitapauksen lopuksi tarkistetaan, onko levykuva luotu eli onko tiedosto olemassa. Tähän voidaan käyttää Pythonin os.path.isfile(tiedostosijainti)-funktiota (kuva 19), jolle annetaan parametrina tiedostosijainti.

```python
if os.path.isfile(workDir +"qnx-bsp-nuc/nuc_bsp/images/usb.img"):
    test.passes("Image file found")
else:
    test.fail("Image not found")
```

KUVA 19. Levykuvan olemassaolon tarkistus

3.3 Kuvan kirjoitus

Levykuvan kirjoitus-testitapauksessa pyritään kirjoittamaan aikaisemmassa testitapauksessa luotu käyttöjärjestelmäkuva SD-MUX-piirin avulla SD-korttiin. Tällä hetkellä fyysinen infrastruktuuri on sellainen, että isäntäkoneessa on kiinni useita

SD-MUX-piirejä. Levykuvan kirjoituksen alustus toteutetaan siten, että kaikki muut piirit irrotetaan tietokoneesta ohjelmallisesti, jotta isäntäkoneelle näkyy vain yksi SD-kortti. Näin ollen on helppo löytää oikea kortti, jolle kirjoittaa.

Levykuvan kirjoitus tapahtuu Linuxin dd-ohjelmalla Shell-skriptissä, jossa on kovakoodattuna osuutena levykuvan sijainti käyttöjärjestelmässä. Muuttuva osa ympäristössä on siirreltävän muistivälineen osoite, sillä se saattaa vaihdella. Linux ympäristössä laitteet näkyvät /dev/-puun alla, jossa siirreltävät muistivälineet näkyvät aakkosjärjestyksessä /dev/sda, /dev/sdb jne. Laitteen kirjoitusosoitetta ei kuitenkaan voida kovakoodata vaikkapa /dev/sdb-osoitteen alle, sillä se ei aina pidä paikkaansa. Laitteet saattavat jäädä “haamuilemaan” ja näin ollen voivat pilata kuvan kirjoituksen. Haamuileminen tarkoittaa sitä, että tietokoneessa näkyy esimerkiksi sdb-levyasema, vaikka se on jo poistettu. Siirreltävän muistivälineen eli SD-kortin osoitteen haku toteutetaankin putkitetulla komennolla, jossa hyödynnetään Linuxin komentoriviohjelmia lsblk, grep ja awk. Näin ollen varmistetaan kuvan kirjoitus oikeaan asemaan (kuva 20).

```
DISK_NAME=$(lsblk -p -S -o NAME,TRAN,SIZE | grep -E '(usb)' | awk {'print $1'})
echo Flashing to $DISK_NAME...
sudo dd if=${IMAGE} of=${DISK_NAME} bs=1M conv=noerror,fsync oflag=direct status=progress
sync
ret=$?
```

KUVA 20. Levykuvan kirjoitus

Levykuvan kirjoituksen jälkeen tietokoneen pitäisi olla valmis testausta varten. Aikaisissa testikokoelman testauksen vaiheissa todettiin kuitenkin, että testattavan tietokoneen Internet-yhteys kaatuu noin kahden minuutin jälkeen tietokoneen käynnistyksestä.

Testilaitteiden automaattiset virtakäynnistykset on automatisoitu releillä. Intel NUC -tietokoneen tapauksessa on laite laitettu käynnistymään automaattisesti, mikäli virta on katkennut äkillisesti. Tämä on niin sanottu Power on after power failure -asetus. Asetuksen saa Intel NUC -koneesta päälle BIOS-valikon kautta. Verkko-ongelma juontaa juurensa luultavasti tästä pakotetusta virransäätelystä, jossa tietokone ei saa itse käydä läpi omaa sammutusprosessiaan. Ongelmaan

mietittiin ratkaisua ja todettiin, että jos tietokone käynnistetään uudelleen turvallisesti komentorivikäskyllä, ongelma korjaantuu. Tämän testitapauksen loppuun laitettiinkin komentorivikäsky ottamaan SSH-yhteydellä laitteelle yhteys ja ajaa komento shutdown -S reboot ennen verkkoyhteyden katkeamista. Tällä ratkaisulla saadaan käskettyä laite uudelleenkäynnistämään itse itsensä turvallisesti. Ratkaisu osoittautui toimivaksi.

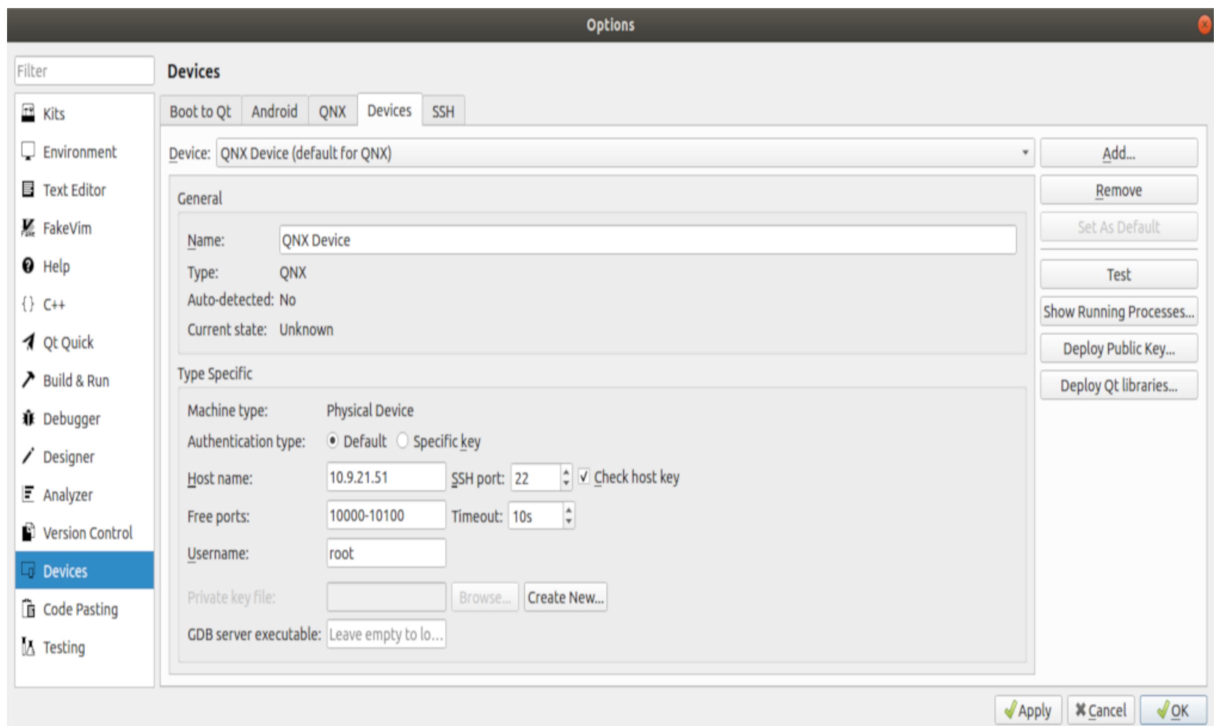
Testikokoelman aikainen testaus oli ratkaisevaa, sillä oikean ympäristön ongelmat eivät olisi muuten ilmaantuneet hyvissä ajoin. Testauksen aikana syntyi myös päätös yleisen SSH-avaimen sisällyttämisestä levykuvaan.

3.4 Squish-serveritiedosto

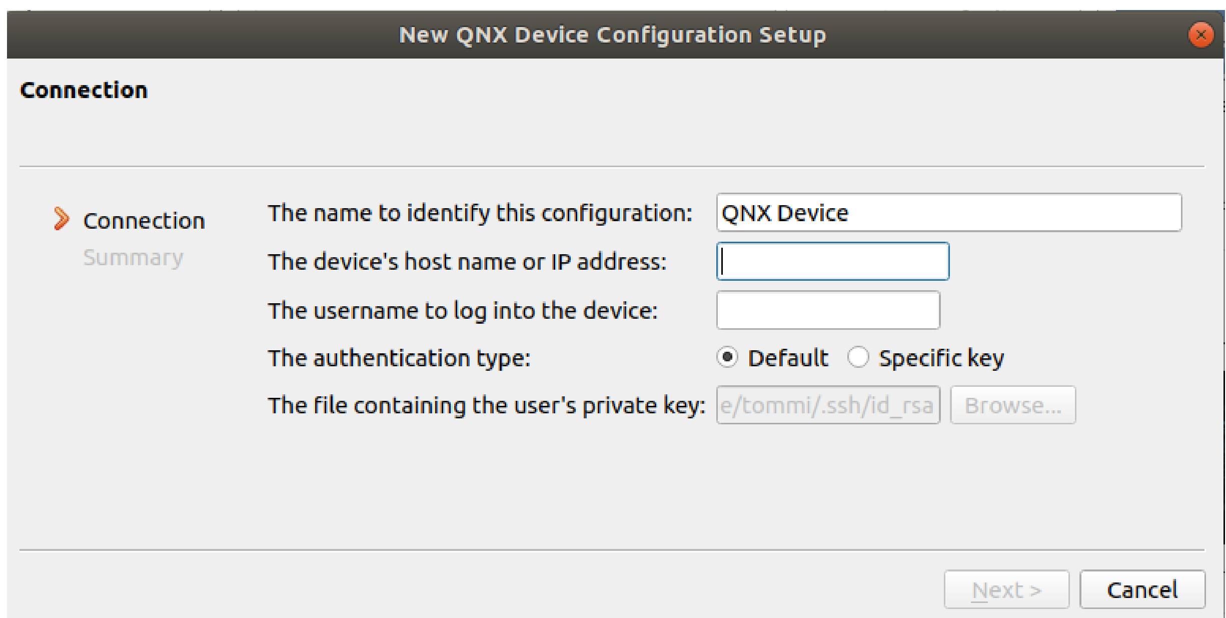
Squish-testausympäristö tarvitsee tiedot sovelluksista, joita se testaa. Tätä varten Squish käyttää server.ini-nimistä tiedostoa, jossa on Creator-ohjelman polku sekä ohjelmien nimet, joihin Squish voi kiinnittyä. Tämä tiedosto pitää poistaa vanhojen testien jäljiltä ja luoda uusi tiedosto uusilla arvoilla. Muutoksia varten tehtiin yleinen funktio, jolle annetaan listaparametrina testattavien ohjelmien nimet listaobjektina, sekä laitteen IP-osoite.

3.5 QNX-laitteen lisäys

Testattava laite pitää lisätä Creatoriin, jotta laitteelle voidaan lähettää binäärit. Laite lisätään IP-osoitteen avulla. Tämä on helppo ja yksinkertainen vaihe toteuttaa, sillä laitteessa tulee olemaan staattinen IP-osoite, joten se tulee pysymään aina samana. Laite lisätään syöttämällä laitteelle nimi, IP-osoite ja käyttäjätunnus. Kuvassa 21 näytetään asetusvalikko, jossa on jo lisättynä laite. Kuva 22 näyttää QNX-laitteenlisäystyökalun.



KUVA 21. Uuden laitteen lisäämisvalikko

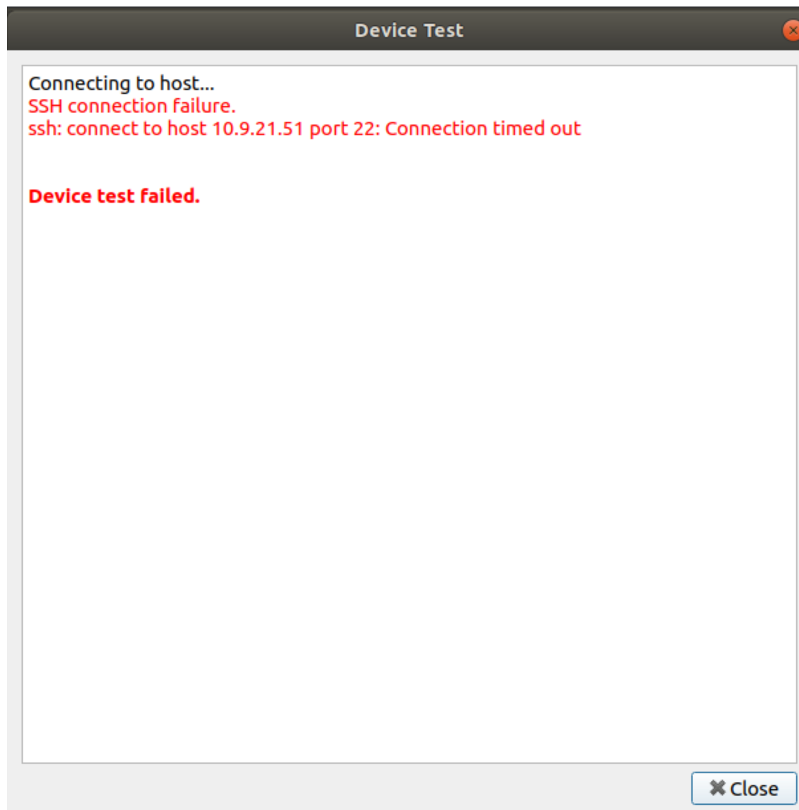


KUVA 22. Laitteen lisäämisvalikko

3.6 Laitteen testaus

Tässä vaiheessa on hyvä testata laitteen toimivuus. Creator tarjoaakin kätevän työkalun laitteen Internet-yhteyden sekä binäärien tarkistukseen. Lisäksi työkalu tarjoaa palautteen, mikäli kaikki on mennyt oikein: "Device test finished successfully”. Tätä voikin hyödyntää kirjoittamalla testitapaukseen if -else -tyyppinen ehtolause. Ehtolauseeseen voi laittaa test.passes() funktion, jolla saa hyvän indikaattorin laitteen toiminnasta.

Tämä testitapaus käynnistää siis Device Test -ohjelman ja odottaa ulostulosyötettä “Device test finished succesfully.” Kuvassa 23 näkyy laitetestityökalun tilanne jossa laitteeseen ei saada yhteyttä.



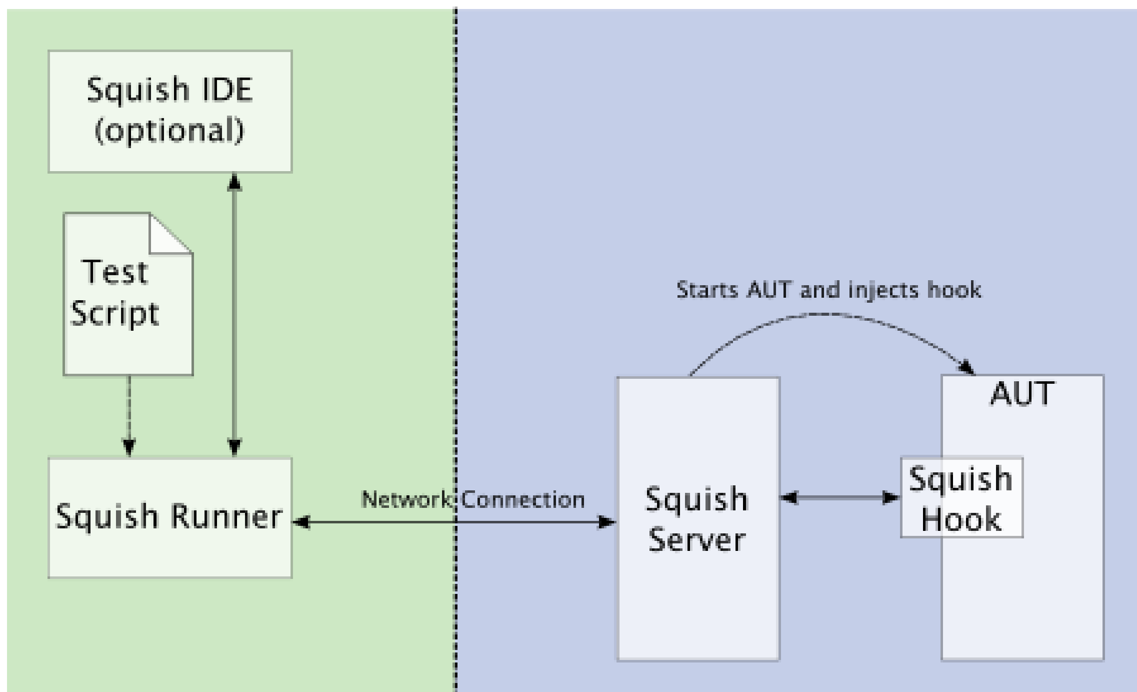
KUVA 23. Testi ei onnistunut

3.7 Testattavan laitteen ympäristön alustus

Testien ajamista varten tulee ympäristö alustaa. Squish-käyttöliittymätesterin kanssa toimittaessa tarvitaan testilaitteeseen Squish-kirjastot, lisäosa (plugin) ja pieni kirjastotiedosto nimeltään koukku (Squish hook). Osuva nimi sinänsä, sillä

sen ainoa tehtävä on kiinnittyä ohjelmaan, jotta Squish saa tietoa ohjelman kulusta.

Koukku ja Squish-serveri kommunikoivat tcp-pistokkeiden avulla ja näin ollen Squish pääsee käsiksi testattavan sovelluksen objekteihin ja näkee niiden tilan (10.1 Squish Concepts). Squish-testaus perustuu testausskriptin ja testattavan sovelluksen (AUT) erillisiin prosesseihin (kuva 24). Tällä arkkitehtuurilla saavutetaan se, että vaikka testattava ohjelma kaatuisikin, ei se kaada testausskriptin ajoa. (10.1 Squish Concepts.)



KUVA 24. Arkkitehtuuri (10.1 Squish Concepts)

Jotta arkkitehtuurin saa toimimaan, pitää siirtää yllämainitut tiedostot siirtää testilaitteeseen ja sen jälkeen kertoa ympäristömuuttujien avulla, mistä kyseiset tiedostot löytyvät. (7.3.3. Setting Environment Variables).

Kirjastojen toimivaksi sijoituspaikaksi osoittautui /qt/lib, eli sijoittaminen Qt:n kirjastojen sekaan. Tämä ratkaisu ei tuntunut aiheuttavan minkäänlaisia ongelmia. Lisäosa sijoitettiin /qt/plugins/generic-tiedostopuun alaisuuteen. Tiedostojen siirtoa varteen luotiin Shell-skripti, joka hyödyntää scp-ohjelmaa tiedostojen siirtoa varten. Skriptiin (kuva 25) annetaan parametreina käytetty Qt:n versio sekä laitteen IP-osoite. Parametrien käyttö skripteissä takaa sulavan toimivuuden, vaikka

versionumerot muuttuisivatkin. Testaavaan tietokoneeseen lisättiin nämä tiedostot polkuun /opt/squish/qnx/5.11.1/intel_nuc

```
scp -o StrictHostKeyChecking=no /opt/squish/qnx/$2/intel_nuc/plugins/generic/libsquishplugin.so
root@$1:/qt/plugins/generic
sleep 1
scp -o StrictHostKeyChecking=no -r /opt/squish/qnx/$2/intel_nuc/squish/lib/. root@$1:/qt/lib
```

KUVA 25. Tiedostojen siirto-skripti

Tiedostojen ollessa paikoillaan pitää ympäristömuuttujien olla asetettu. Muuttujien asetukseen käytettiin hyödyksi Creatorin omaa Run Environment -asetusta (kuva 26), jolla voidaan suorittaa ympäristömuuttujien asetus.

Laitteeseen laitettiin kaksi ympäristömuuttujaa: QT_QPA_GENERIC_PLUGINS sekä SQUISH_PREFIX. Nämä ympäristömuuttujat antavat Qt:lle tiedon ladata Squish-lisäosan sekä tiedon siitä, mistä polusta Squish -työkalun tarvitsemat tiedostot löytyvät.

Run Environment

Use **System Environment** and
Set QT_QPA_GENERIC_PLUGINS to **squish:10100**
Set SQUISH_PREFIX to **/qt/**

Details

Base environment for this run configuration: System Environment

Fetch Device Environment

Variable	Value
QT_Q...	squish:10100
SQUI...	/qt/

Edit
Add
Reset
Unset
Batch Edit...
Open Terminal

KUVA 26. Ympäristömuuttujat

Squish-hook eli koukku on itse käännettävä tiedosto, jonka binäärien kääntämiseen tarvitaan Squish-lähdekoodi ja valmis Qt:n asennus. Koukun kääntämistä

ei tarvitse testeissä suorittaa eli se täytyy tehdä vain kerran ja tallentaa testaavan tietokoneen tiedostoihin, josta se voidaan testin yhteydessä hakea. Testiympäristössä koukku tallennetaan /opt/squish-kansioon. Squish-koukku luodaan kuvan 27 mukaisesti.

```
./configure --enable-qmake-config --disable-all --enable-qt --enable-server
--with-squishidl=<squishidl binäärin sijainti> --with-qmake=<qmake binäärin sijainti>

./build

./build install DESTDIR=<sijainti minne tallennetaan>
```

KUVA 27. Squish-koukun luominen

Jotta koukku toimii testauksessa, se pitää lähettää laitteelle projektin mukana. Koukku saadaan lisättyä laitteeseen sisällyttämällä se projektin .pro-päätteiseen tiedostoon. Kätevin tapa toteuttaa tämä on sisällyttää koukku ensimmäisen testattavan applikaation .pro-tiedostoon (kuva 28), jonka jälkeen koukku on paikoillaan.

```
TEMPLATE = app
TARGET = wearable
QT += quick quickcontrols2

SOURCES += \
    wearable.cpp

RESOURCES += \
    wearable.qrc

target.path = $$[QT_INSTALL_EXAMPLES]/quickcontrols2/wearable
INSTALLS += target
include(/opt/squish/qnx/5.11.1/intel_nuc/qtbuiltinhook.pri)
```

KUVA 28. Koukun lisäys .pro-tiedostoon

3.8 Sovellustestaus

Tässä osassa käsitellään sovellustestit ja tavoite, mitä näillä testeillä haetaan. Testikokoelmassa on alustavasti neljä sovellustestiä, joista kaksi on esimerkkiapplikaatioiden käännös- ja lähetystestejä. Loput kaksi sovellustestiä ovat tyhjän Qt-sovellusprojektin luomiseen liittyviä. Tyhjinä sovellusprojekteina toimivat

Qt Quick-sekä Qt Widget -sovellustyypit. Näin ollen saadaan testattua, että ainakin osa esimerkkiapplikaatioista toimii sekä tyhjän projektin luonti ja ajo onnistuu.

Esimerkkiapplikaatio on Creatorissa valmiiksi mukana tuleva esittelysovellus, jolla voidaan esitellä jonkin Qt:n moduulin toimintaa. Esittely voi liittyä vaikkapa 3D-kuvan piirtämiseen, taulukkorakenteisiin, verkkoyhteyksiin jne. Qt tarjoaa sovelluskehityksen tueksi useita erilaisia moduuleita, joilla voidaan saada aikaan erinäisiä toiminnallisuuksia. Esimerkkiapplikaatiot ovat tapa esitellä näitä toiminnallisuuksia.

Esimerkkiapplikaatioiden testauksen arkkitehtuuriin kuuluu testattavan sovelluksen valinta, kääntäminen, lähettäminen ja testaus. Testausprosessi on hyvin samantyyppinen kaikissa esimerkkisovellustesteissä, joten prosessi kannattaa tallentaa funktioiksi ja antaa funktion argumenttina testattava sovellus. Tämäntyyppinen funktio löytyy jo valmiiksi kirjoitettuna yrityksen testiautomaatioskripteistä, mutta sitä pitää muokata tämän työn tarpeisiin. Esimerkki tämänkaltaisesta tarpeesta on, että jokaisessa sovelluksen käynnistystestissä tulee kertoa QNX-käyttöjärjestelmälle mistä hakea Qt:n kirjastot.

Edellisessä luvussa puhuttiin ympäristömuuttujien asettamisesta Creatorin asetusvalikon avulla. Tämä käy niin, että valmiina olevaan funktioon laitetaan koodin väliin sopivaan kohtaan skripti, joka asettaa muuttujat. Ympäristömuuttujien lisäksi tulee funktioon laittaa vaihtoehtoinen sovelluksen pysäytystapa. Muokkaus liittyy siihen, että sovelluksen pysäytysnappi ei toimi QNX-sovellusten kanssa Qt:n versiossa 5.11.1. Testauskokoelma tehtiin tuon version aikana, joten ratkaisuna käytettiin SSH-yhteyden yli käytettävää slay <applikaatio> -komentoa kuvan 29 mukaisesti.

```
'ssh -t -o StrictHostKeyChecking=no root@' + device_ip +  ' /bin/sh -l -c "slay\
' + appname_lowercase + '"'
```

KUVA 29. Slay-komento

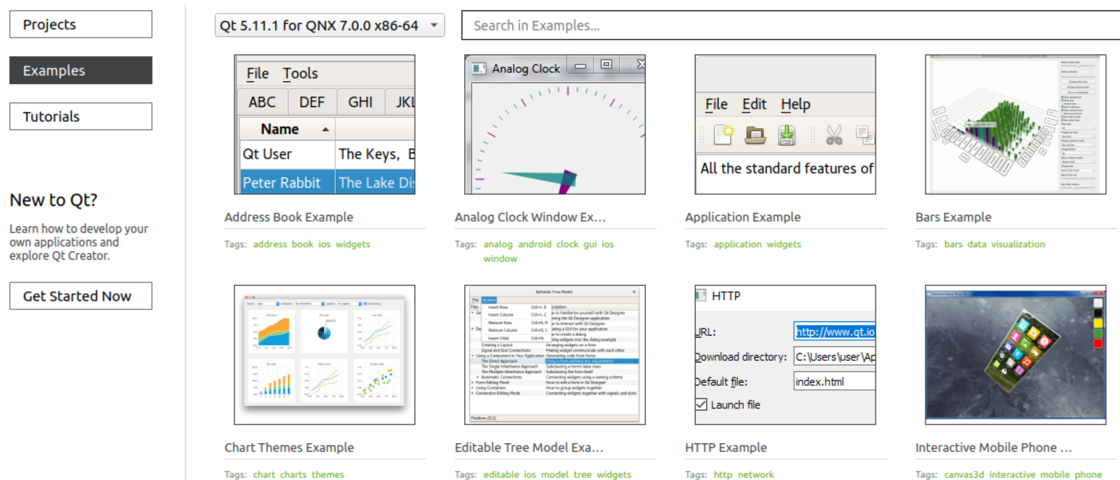
Havaittiin, että komennossa pitää antaa ohjausmerkki sanan “slay” jälkeen, jotta välilyönti rekisteröidään komennossa oikein. Ilman ohjausmerkkiä “\” komento antaa virheilmoituksen.

Isäntäkoneen ssh -o valinta tarkoittaa option lisäämistä, eli se on StrictHost-KeyChecking=no parametrin mahdollistava lippu. Avaimentarkistusoption poistaminen mahdollistaa sen, että yhteydenotto toimii testattavan laitteen SSH-avaimen muutoksista huolimatta. Option poistaminen antaa Man in the middle -tyyppisen hyökkäyksen varoituksen, mutta ratkaisu on turvallinen yrityksen sisäisessä testiverkossa työskenneltäessä.

Kohdelaitteen Shell-istunnolle annettiin komennon parametreiksi -l sekä -c. Parametri -l tarkoittaa Login Shell -istuntoa. Jälkimmäinen parametri -c tarkoittaa komentosanan suorittamista, joka on slay-komento. (sh(1) - Linux man page).

3.9 Sovellustestaustyypit

Esimerkkisovellusten testauksella haetaan sitä, että saadaan testattua esimerkkisovelluksen lähdekoodin kääntäminen kohdelaitteen binääreiksi, eli konekielelle sekä sovelluksen toimivuus kohdelaitteessa.



KUVA 30. Esimerkkisovelluksia

Testikokoelman kaksi muuta sovellustestiä ovat testejä, joissa luodaan kaksi erityyppistä tyhjää Qt-sovellusprojektia ja testataan, toimiiko tavallinen Creatorin käyttö. Esimerkkisovellustestien tavoin projektit käännetään ja lähetetään laitteelle. Tyhjille projekteille ei ole tehty esimerkkisovellusten tyylisiä testejä, sillä ne ovat vain tyhjiä projekteja, eikä niissä ole klikattavia objekteja.

Edellä mainitut tyhjän projektin luomistestit ovat tärkeitä, sillä ne ovat lähimpänä sitä, mitä loppukäyttäjän kehitystyön rutiinit ovat. Testaamisessa pitää pyrkiä testaamaan asioita, jotka ovat tavanomaisimpia loppukäyttäjän käyttämiä ja tekemiä asioita.

3.10 Vianetsintätyökalun testaus

Kehittäjän tärkeä apu sovellusten kehityksessä ja virheenkorjauksessa on niin sanottu debuggeri. Työkalun avulla voidaan luoda pysäytyspisteitä koodiriveille ja näiden pisteiden avulla pysäyttää sovelluksen eteneminen. Pysäytyspisteitä käytetään sovelluksen sen hetkisen tilan tutkimiseksi.

Virheen etsinnän testaaminen keskittyy C++ -ja QML-lähdekoodin pysäytyspisteiden toimivuuden tarkistamiseen. Testaaminen suoritetaan yhdessä testitapauksessa. Testissä avataan esimerkkisovellus ja siihen lisätään C++-pysäytyspiste ja QML-pysäytyspiste. Ensin testataan koodin pysähtyminen QML-pysäytyspisteellä ja sen jälkeen testataan C++-pysäytyspiste.

Pysäytyspisteen sijoittaminen suoritetaan for-silmukalla, joka käy lähdekoodia läpi rivi kerrallaan palauttaen rivinumeron, jolla tällä hetkellä on. Rivinumeroa verrataan muuttujaan, joka on ennen luuppia annettu haluttu pysäytyspisteen rivinumero. Silmukan ollessa oikealla rivinumerolla suoritetaan nativeType()-funktiolla F5-näppäimen painallus, joka aktivoi pysäytyspisteen aktiiviselle riville. For-silmukkaa käytetään molempien lähdekoodien pysäytyspisteiden asetukseen.

Testaamisen onnistumisen indikaattorina käytetään vianetsintätyökalun ulostulosyötettä, josta etsitään koodin pysähtymisestä indikoivaa viestiä (kuva 31).

```
if waitForObjectExists(names.toolbar_Running_Utils_StatusLabel):
    test.passes("Debugger stopped on QML breakpoint..")
else:
    test.warning("Something went wrong with QML Debugging")
```

KUVA 31. Vianetsintä

Testitapauksessa QML-koodin ja C++-koodin vianetsintätyökalun testaus suoritetaan testitapauksen sisällä kahtena eri testinä, sillä näitä kahta ohjelmointikieltä ei voi ajaa virheenjäljitystilassa samanaikaisesti.

3.11 Testiympäristön alustus

Testauksessa ja varsinkin automaattisessa testauksessa on tärkeää testauksen aloittaminen alustetussa ympäristössä. Tämän takia testikokoelman loppuun sijoitetaan testiympäristön alustukseen ja puhdistukseen tarkoitettuja toimenpiteitä.

Ympäristön alustus suoritetaan siten, että testitapauksessa ajetaan yleinen uninstall_qt()-skripti, joka poistaa Qt:n asennuksen ylläpitotyökalulla. Kun Qt on poistettu, suoritetaan väliaikaisen testikansion ("/home/RTA") poistaminen sisältöineen. Toiminto suoritetaan shutil.rmtree(<dir>)-funktiolla. Hyvä tapa on myös sammuttaa testattava laite, joten testiympäristön viimeiseksi koodinpätkäksi laitetaankin SD-MUX-piirille laitteensammutuskäsky. Testiympäristö on nyt valmis seuraavia testejä varten.

3.12 Jenkins-integraatio

Tässä työssä ei sukelleta syvälle siihen, miten Jenkins toimii, vaan selitetään pintapuolisesti toimenpiteet, joilla saatiin testikokoelma Jenkinsiin.

Työn saaminen Jenkinsiin vaati oman työnimen, joka identifioi työn. Tämä vaihe toteutettiin antamalla työlle ensin järkevä työnimi. Kun työnimi oli annettu, lisättiin se Jenkinsin konfiguraatioiden julkaisuarkistoon. Työnimen lisäksi työkonfiguraatioon laitettiin tieto siitä, missä testikokonaisuus sijaitsee. Työ lisättiin Jenkinsiin valmiin työpohjan avulla, joten siirto oli nopea ja vaivaton. Työpohja on työn asetukset sisältävä tiedosto.

Toinen työvaihe oli nimikkeen luominen työlle. Nimikkeen avulla Jenkins osaa ohjata työn oikealle koneelle. Tämä on tärkeää, sillä testattava laite on kiinni ainoastaan tietyssä koneessa.

Jenkinsin integroiminen työhön oli nopea operaatio, sillä Jenkins integraation yhteydessä ei luotu uusia arkkitehtuureja vaan käytettiin käytössä olevaa. Käytännössä koodirivilisäykset olivat tässä tapauksessa vähäiset.

3.13 Toimivan kokonaisuuden testaus

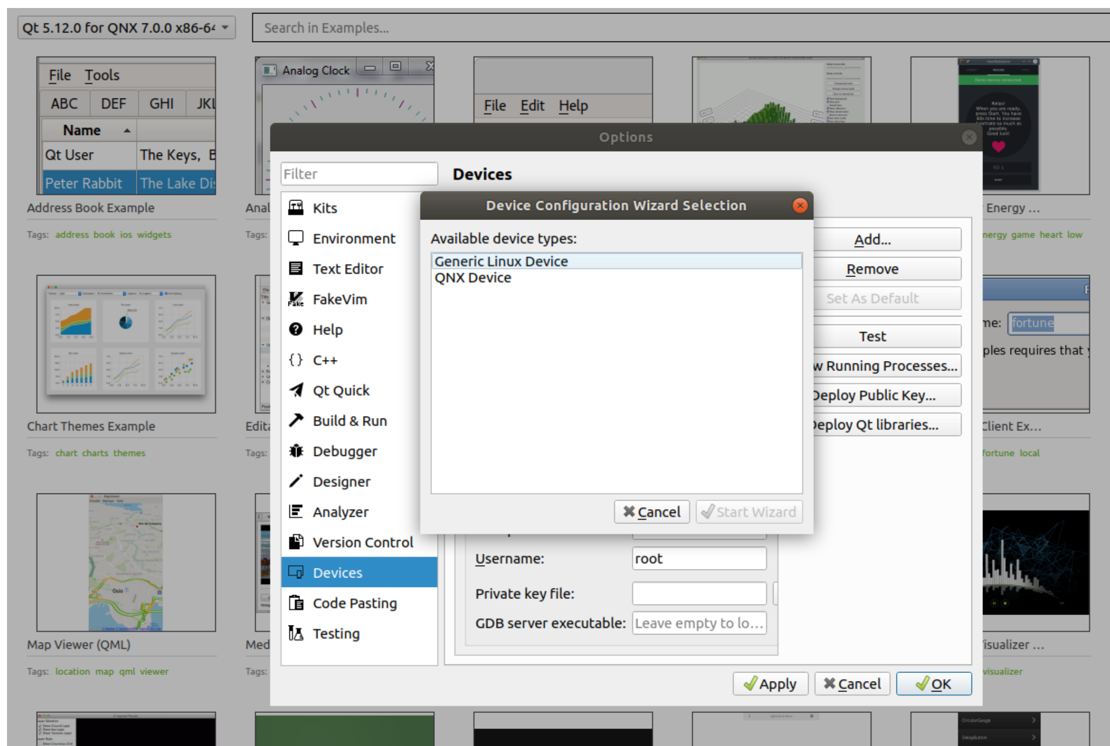
Työn ollessa testausta vaille valmis oli se syytä testata. Työkokonaisuuteen oli kulunut useampi kuukausi ja kokonaisuus alkoi olemaan kasassa. Oli aika testata, miten testit toimivat Jenkinsin kautta ajettuna, sillä palapelin viimeinen pala eli Jenkins-integraatio oli kasassa.

Testauksessa on vielä oma lisänsä se, että vaikka testausskripti toimii paikallisesti testattuna, se ei välttämättä toimi Jenkinsin kautta ajettuna. Työn rakennus onnistui kuitenkin niin hyvin, että siirrettäessä työtä varsinaiseen testausympäristöön näitä ongelmia ei ilmentynyt kuin yksi.

Tämä harmaita hiuksia aiheuttanut ongelma, joka toistui ainoastaan Jenkinsin kautta ajettuna liittyi siihen, että Squish ei osannut pitää aktiivisena ikkunana ylintä näkemäänsä ikkunaa. Tämä johti siihen, että testin alussa skripti ei osannut klikata QNX-laite -valintaa optiovalikosta. Epäonnistuneen yrityksen jälkeen kaikki testit epäonnistuivat, koska testikokonaisuudessa ei ollut testattavaa laitetta. Ongelma liittyi siis ikkunan tuomiseen etualalle, johon onneksi oli Froglogicilla kehitetty jo ratkaisu. Ratkaisu oli ikkunan kohottaminen ja aktivoiminen (kuva 32). (Problem - Bringing window to foreground (Qt).)

```
deviceWindowDict = {"name": "listWidget", "type": "QListWidget", "visible": 1}
deviceWindow = waitForObject(deviceWindowDict)
button = waitForObjectItem(names.projectExplorer_Internal_DeviceFactorySelectionDialog\
_listWidget_QListWidget, "QNX Device")
deviceWindow.show()
getattr(deviceWindow, "raise")()
deviceWindow.activateWindow()
mouseClick((button), 1, 1, Qt.NoModifier, Qt.LeftButton)
snooze(2)
```

KUVA 32. Ongelmaan ratkaisu



KUVA 33. Ongelmatilanteen luonut sovellusikkuna.

Muuta huomionarvoista oli se, että työn aikana ehti tulla myös uusi LTS-julkaisu 5.12.0. Tämä johti siihen, että työtä piti loppuvaiheessa alkaa muokkaamaan tukemaan mahdollisia uusia muutoksia. Tämä ei sinänsä ollut hankala tehtävä, sillä työ oli rakennettu dynaamisten versionumerojen varaan.

4 YHTEENVETO

Lopputuloksena tästä opinnäytetyöstä yritys sai käyttöönsä mielestäni todella hyvin toimivan testikokonaisuuden. Tästä työstä oli siis ihan konkreettista hyötyä. Testauskokonaisuuden avulla yrityksessä voidaan nyt automaattisesti testata Qt:n toimivuus QNX-reaaliaikakäyttöjärjestelmässä. Ajallisesti testikokoelman ajaminen kestää noin 18 minuuttia.

Alkuvaiheessa mainittua testien automaattista liipaisemista Jenkinsistä ei tähän testikonfiguraation lisätty. Mikäli tälle ominaisuudelle tarvetta myöhemmin ilmenee, se lisätään.

Työn aikana tuli ilmeiseksi, että vanha testausarkkitehtuuri saa luvan väistyä. Uuden arkkitehtuurin suunnittelu ei tule vaikuttamaan tämän opinnäytetyön tuotoksiin negatiivisesti, sillä rakennettuja skriptejä on helppo jatkojalostaa. Työni tulee luultavammin jatkumaan vielä aiheen parissa. Työn suoritus onnistui niin hyvin, että se valmistui hyvissä ajoin ennen työsuhteen loppua. Näin ollen minulla jäi aikaa tehdä muita mielenkiintoisia projekteja, kuten QBSP-pakettien automaattinen testaus sekä laitteidenhallintajärjestelmän implementointi tulevaisuuden testausympäristöön.

Työstä viisastuneena ja oppineena haluan painottaa aikaisen testauksen merkitystä aikataulutuksen ja laadunvarmennuksen merkittävänä tekijänä. Mitä aikaisemmin testataan ja huomataan perustavanlaatuisia vikoja tai pullonkauloja, sitä aikaisemmassa vaiheessa voidaan arkkitehtuuria muovata toimivammaksi.

Työ oli mielenkiintoinen ja pääsin oppimaan sen aikana uusia asioita. Koen, että minusta on tullut sekä parempi ohjelmoija, että parempi testaaja tämän työn jäljiltä. Työn kirjoittamiseen meni monta syksyn pimeää iltaa. Työn valmistuminen kuitenkin venyi ja venyi, joten sain lopulta viimeistellä tämän työn kevätauringon paisteessa. Ei huono sekään.

LÄHTEET

7.3.3. Setting Environment Variables. Froglogic. Saatavissa: https://doc.froglogic.com/squish/latest/rg-autsettings.html#rgas-envars. Hakupäivä 23.12.2018.

1. 8. Errors and Exceptions. 2019. The Python Tutorial. Saatavissa: https://docs.python.org/2.7/tutorial/errors.html. Hakupäivä 20.9.2018.

10.1 Squish Concepts. Froglogic. Saatavissa: https://doc.froglogic.com/squish/5.1/tgs-concepts-mac.html. Hakupäivä 1.11.2018.

About Qt. 2018. Qt Wiki. Saatavissa https://wiki.qt.io/About_Qt. Hakupäivä 1.10.2018.

About Us. 2018. The Qt Company. Saatavissa: https://www.qt.io/company. Hakupäivä 12.11.2018.

Company. Froglogic. Saatavissa: https://www.froglogic.com/company/. Hakupäivä 21.1.2019.

Expect(1) - Linux man page. Die.net. Saatavissa: https://linux.die.net/man/1/expect. Hakupäivä 10.10.2018.

Intel NUC. Newegg. Nettikauppa. https://www.newegg.com/Product/Product.aspx?Item=N82E16856102184. Hakupäivä 16.3.2019.

Kawaguchi, Kohsuke - Neale, Michaels. 2018. Distributed builds. Jenkins Wiki. Saatavissa: https://wiki.jenkins.io/display/JENKINS/Distributed+builds. Hakupäivä 12.11.2018.

Multiple Real-World Scripting Languages. Froglogic Saatavissa: https://www.froglogic.com/squish/features/multiple-real-world-scripting-languages/. Hakupäivä 21.1.2019.

Problem - Bringing window to foreground (Qt) Froglogic. Saatavissa: https://kb.froglogic.com/display/KB/Problem+-+Bringing+window+to+foreground+%28Qt%29. Hakupäivä 19.1.2019.

SD MUX. 2018. Saatavissa: https://wiki.tizen.org/SD_MUX. Hakupäivä 10.10.2018

Software release life cycle. 2019. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Software_release_life_cycle. Hakupäivä 20.1.2019

Pettichord, Bret 2001. Seven Steps to Test Automation. Saatavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.9363&rep=rep1&type=pdf. Hakupäivä 27.1.2019.

Pettichord, Bret 1996. Success with test automation. Saatavissa: http://ceng557.cankaya.edu.tr/uploads/files/Success%20with%20Test%20Automation.doc. Hakupäivä 9.10.2018.

Test Automation. 2018. Saatavissa: https://en.wikipedia.org/wiki/Test_automation. Hakupäivä 27.1.2019

QNX 7.0 on Intel NUC. 2018. Sisäinen dokumentti. The Qt Company.

Qt Quality Management. 2018. Sisäinen dokumentti. The Qt Company.