Tomi Jolkkonen

Kriittisten järjestelmien ohjelmointi

Tietotekniikan kandidaatintutkielma

13. joulukuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Tomi Jolkkonen

Yhteystiedot: tomi.j.jolkkonen@student.jyu.fi

Ohjaaja: Sanna Mönkölä

Työn nimi: Kriittisten järjestelmien ohjelmointi

Title in English: Safety-Critical Systems Programming

Työ: Kandidaatintutkielma

Sivumäärä: 27+0

Tiivistelmä: Kriittiset järjestelmät teho-ostastoilla, ydinvoimaloissa, avaruudessa ja muissa, jopa vihamielisissä ympäristöissä, suunnitellaan selviytymään yllättävistäkin tilanteista. Laitteet tai ohjelmat eivät tällaisissa paikoissa voi kaatua ongelmatapauksissa, tai siitä voi seurata hengenvaarallisia ketjureaktioita. Tällaisten järjestelmien, laitteiden ja ohjelmien suunnittelu on eittämättä tarkkaa työtä ja sisältää monia hyväksi todettuja käytäntöjä, joita voisi hyödyntää muissakin asiayhteyksissä. Tämä tutkielma ottaa kantaa tähän laajaan aihepiiriin rajatun kirjallisuuskatsauksen avulla. Tavoitteena on ymmärtää paremmin ohjelmistoja ja ohjelmointia kriittisissä järjestelmissä sekä niihin liittyviä yleisimpiä käytäntöjä, virheitä ja ohjeistuksia. Tuloksena on yhteenveto pääkohdista, joilla ohjelmointityötä voi kehittää paremmaksi kaikilla toimialoilla. Lisää tutkimusta voisi tehdä toimialakohtaisesta ohjelmoinnista, koska eri aloilla (vrt. lääketiede vs. pelit) on omat hyväksi todetut käytäntönsä, jotka eroavat toisistaan.

Avainsanat: Kriittiset järjestelmät, Ohjelmointi, Parhaat käytännöt

Abstract: Safety-critical systems in hospitals, nuclear power plants, space programs and other hostile enviroments are developed to survive in surprising problematic situations. In these kinds of environments, hardware and software cannot crash down without creating life-compromising chain reactions. Developing safety-critical systems, machines and programs is propably a very hard and detailed work which includes best practises that we can learn from and use in other industries too. This

study uses scientific bibliography to find out and better understand how we should develop our programming habits to create programs with better quality and fewer errors. As a summary, a list of ideas is introduced that could be used in any project in order to create programs with higher quality in any industry. More studies could be done by specific areas of industries, for there are differences in best practises depending whether you're creating a pace maker or a video game.

Keywords: Safety-Critical Systems, Programming, Best Practises

Kuviot

Kuvio 1. Kääntäjäverifikointi (Avinash Brown 2015). 9
Kuvio 2. Auto on esimerkki yhdistelmästä ohjelmia ja laitteiden ohjelmointia. Kuvassa autonkaappausohjelma (Koscher et. al 2010). 17

Sisältö

1 JOHDANTO 1

2 OHJELMOINTITAITO 3
2.1 Yleisimmät ohjelmointivirheet 3
2.2 NASAn 10 käskyä hyvään koodiin 4

3 KEHITTYMINEN KRIITTISEN JÄRJESTELMÄN OHJELMOIJAKSI 5
3.1 Mikä on kriittinen järjestelmä 5
3.2 Kriittisten järjestelmien jaottelu 7

4 KRIITTISEN JÄRJESTELMÄN OHJELMOINTIHAASTEITA 8
4.1 Kustannustehokkuus 8
4.2 Pelkkä lähdekoodi ei riitä 9
4.3 Puutteelliset työkalut 10
4.4 Konservatiiviset standardit eivät vastaa nykypäivää 11
4.5 Ohjelmointikielten riittävä määrittely 12
4.6 Haastava fyysinen ympäristö 13

5 KOHTI JÄRJESTELMIEN AMMATTIMAISTA OHJELMOINTIA 14
5.1 Ohjelmointikielen rajattu käyttö 14
5.2 Apua matematiikasta 15
5.3 Myös laitteet tulee ohjelmoida 16
5.4 Seppä ja kisälli - Opi tekemällä 17

6 YHTEENVETO 19

KIRJALLISUUTTA 20

1 Johdanto

Tässä rajatussa kirjallisuuskatsauksessa käydään läpi lähteiden avulla kriittisten järjestelmien ohjelmointia sekä ohjelmoinnin että ohjelmointiympäristön kannalta. Alussa määritellään kriittinen järjestelmä (engl. *safety-critical system*), joka on järjestelmä, jonka kaatuminen saattaa aiheuttaa ihmishenkien menetyksiä, suurta omaisuusvahinkoa tai vahinkoa ympäristölle (Knight, 2002).

Tämän jälkeen käydään läpi yleisemmin kriittisen järjestelmän ohjelmointihaasteita. Vaikka nykyään on runsaasti tietoa ja standardeja, ei ole yhteistä sopimusta siitä, miten kriittisiä järjestelmiä tulisi ohjelmoida ilman, että kustannukset nousevat kohtuuttomiksi (Wong et al. 2011). Ohjelmoinnissa tulee kiinnittää huomioita muuhunkin kuin korkean tason koodiin, kuten objektikoodin validointiin (Avinash, 2015) ja työkaluihin (Axelrod, 2014). Standardien päivittäminen tähän päivään on yksi konservatiivisten julkisten virastojen haaste, koska vanhoissa standardeissa ei aina puhuta ohjelmistoturvallisuudesta (Bernsmed et al. 2018). Yksi ohjelmistoturvallisuuden ydinkysymys kriittisten järjestelmien ohjelmoinnissa onkin ohjelmointikielen määrittely. Kun tehdään ohjelmistoja ympäristöissä, joissa hengenvaara on läsnä, ohjelmointikieli tulee määritellä hyvin ja sen tulee kestää monenlaisia haasteita (Cullyer, 1991). Siksi kriittisissä järjestelmissä suositaan vuosikymmeniä testattuja ja hyväksi todettuja ohjelmointikieliä. Kaupallisella puolella järjestelmät eivät jousta niin paljon kuin viranomaiset haluaisivat (Delic et al. 2015), jonka vuoksi järjestelmien tai kielten vaihtaminen voi olla pitkäaikainen projekti. Yksi tulevaisuuden haasteista on yhdistää kaupalliset intressit teknologioiden, standardien ja mittareiden kanssa.

Keskeinen haaste on myös fyysinen ympäristö, koska kriittiset järjestelmät sijaitsevat usein ydinvoimaloissa, meren pohjassa, avaruudessa, armeijassa tai esimerkiksi teho-osastolla. Fahey (2016) kuvaaa ohjelmistoarkkitehtuuria armeijan miehittämättömässä sukellusveneen etsintälaitteessa, joka jo otsikkotasolla herättää kysymyksen siitä, miten paljon asioita voi mennä pieleen, jos ohjelmisto ei ole hyvin suunniteltu.

Lopuksi keskitytään ammattimaiseen ohjelmointiin kaikkialla, myös kriittisissä järjestelmissä. Tutkielmassa käydään läpi ohjelmointikielten ominaisuuksia ja niiden rajattuja versioita (HaighLandwehr 2015), matematiikkaa apukeinona (Hoare 2015) ja keskittymistä laatuun, ei määrään. Huomataan myös, että laitteet tulee ohjelmoida toimimaan loogisesti ja oikein. Hieman filosofisemmasta kulmasta on myös hyvä miettiä, että kehitys tapahtuu oikeaan suuntaan. On tärkeä varmistaa, että uudet järjestelmät ovat turvallisia, sillä samoja tekniikoita voidaan käyttää sekä hyvään että pahaan, kuten itseohjautuvan auton kaappaamiseen (Pattabiraman et al. 2008).

Ohjelmointia käytetään erilaisten menetelmien toteuttamiseen. Eräs vaihtoehto sekä laadukkaalle että turvalliselle ohjelmistolle voi löytyä geneettisen ohjelmoinnin kautta, jossa haastetaan koko ohjelmoinnin ajattelun mallia. Luonnossa geneettisessä populaatiossa yksilöitä verrataan toisiinsa ja vahvimmat säilyvät elossa. Tämä ohjelmointitapa on jalostettavissa esimerkiksi tekoälyn kanssa, sekä missä tahansa ohjelmointiprojektissa, jossa tehdään suuri määrä ohjelmia, mutta suoritukseen jätetään vain parhaat vaihtoehdot (Pinheiro Schirru 2019).

Lopuksi esitellään Seppä ja kisälli -tapa, jossa opitaan ammattiohjelmoijaksi kokeneemman kollegan siivellä (Vihavainen et al. 2011). Jatkotutkimuksen kohteeksi ehdotetaan monitieteistä tutkimusta siitä, miten eri toimialat ja ympäristöt toimivat yhteisten standardien alla. Koska ympäristöt ja ihmiset vaihtelevat suuresti, on aiheellista kysyä, tulisiko joka toimialalla olla omat standardinsa. Kaikkeen ohjelmointityöhön on yleistettävissä hyviä käytäntöjä, mutta erityisesti kriittisissä järjestelmissä ympäristömuuttujien vaativuus ja monimutkaisuus antaa aihetta pohtia toimialakohtaisen tutkimuksen hyötyjä ohjelmointityöstä, ottaen huomioon myös ihmisten erilaisuus, käyttäytymispsykologia sekä stressitilanteet. Tällainen lähestyminen voisi avata uusia keinoja tehdä laadukasta ja tietoturvallista ohjelmointityötä.

2 Ohjelmointitaito

Kaikki ohjelmointityö, myös kriittisissä järjestelmissä, lähtee hyvästä ohjelmointitaidosta. Ohjelmointia voi harrastaa kotona tai opiskella yliopistossa. On tausta mikä vain, on hyvä tunnistaa yleisimmät virheet sekä hyvät käytännöt, joilla tehdään ohjelmia, on kyse sitten peleistä, laitteista tai ydinvoimaloista. Hyvä koodi on selkeää ja se ottaa huomioon virheet, muuttuvat tilanteet sekä sen, että koodia käsittelee ja lukee iso joukko erilaisia ihmisiä. Jo pienillä muutoksilla saadaan suuri osa ohjelmointivirheistä korjattua.

2.1 Yleisimmät ohjelmointivirheet

Koska kriittisen järjestelmän ohjelmointi ei eroa normaalista laadukkaasta ohjelmoinnista, voidaan perustellusti sanoa, että ohjelmointiosaaminen kaikkiin järjestelmiin lähtee ensimmäisestä ohjelmoinnin peruskurssista. Altadmri ja Brown (2015) tutkivat 37 miljoonaa kääntäjän tekemää käännöstä 250 000 opiskelijan tekemänä ympäri maailmaa yhden vuoden ajalta ja heidän tekemiään yleisimpiä virheitä sekä niiden korjaamiseen kuluvaa aikaa. Tutkimus tehtiin käymällä läpi Java-ohjelmoinnin peruskurssi sekä jatkokurssi, joissa tutkittiin kääntäjien antamia virheilmoituksia.

Mielenkiintoisin anti tutkimuksesta oli se, että syntaktisten virheiden määrä laskee, kun ohjelmointikokemus karttuu, mutta semanttisten virheiden määrä taas kasvaa. Tästä voidaan vetää johtopäätös, jossa ohjelmoijat tekevät huolimattomuusvirheitä tai ovat haluttomia testaamaan ohjelmiaan. Myös uusien asioiden oppimiskynnys saattaa kasvaa. Ohjelmoinnin alkutaipaleella syntaktisia virheitä tulee useammin, mikä on ymmärrettävää, koska opetellaan kokonaan uutta asiaa eli ohjelmointia. Jälkimmäistä ongelmaa eli semanttisten virheiden kasvua on vaikeampi selittää. Toisin sanoen, jos koodissa olevat virheet syntyvät jo näin varhain, on syytä olettaa, että samoja ongelmia löytyy kaikkialta, sekä kriittisistä että ei-kriittisistä järjestelmistä. Apua on tarjolla monelta suunnalta, sillä standardeja turvallisen ohjelmoinnin suorittamiseen löytyy internetistä (ks. esim vuoden 1994 IEEE Standard for Software

Safety Plans, IEEE Std 1228-1994).

2.2 NASAn 10 käskyä hyvään koodiin

Ammattimaiset ohjelmistonkehitysprojektit käyttävät määriteltyjä koodausohjeita. Ne ovat perussääntöjä, jotka kertovat miten ohjelma kirjoitetaan, rakennetaan sekä mitä kielen osia käytetään ja mitä ei (Holzmann, 2006). Yllättävästi, tällä hetkellä ei ole olemassa yhtenäistä konsensusta siitä, mitkä hyvät koodaustavat ovat. Tästä johtuen monet yritysten ohjeistukset voivat olla yli 100-sivuisia ja sisältö vaihtelee henkilökohtaisten tyylitoiveiden ja tarpeellisten tietoturvaohjeistusten välillä. Myös automaattiset tarkistustyökalut eivät aina kuulu ohjeistukseen, vaikka ne ovat ohjelmointityön perustyökalustoa. Ohjeistuksessa on kyse paljon muustakin kuin itse ohjelman koodiriveistä, koska ohjelmat harvoin ovat eristyksissä muusta maailmasta.

NASA on tehnyt 10 kohdan listan, jonka lukeminen ja noudattaminen on yksi vaihtoehto ammattimaiseen ohjelmointiin. Listaa voi käyttää ohjenuorana mihin tahansa ohjelmointityöhön kriittisten järjestelmien lisäksi. NASA käyttää paljon C-ohjelmointikieltä, mutta ohjeet soveltuvat kaikkeen ohjelmointiin (Holzmann, 2006). Ensiaskelia hyvälle koodille ovat mm. tekninen ja tiedon tuki, vahvat lähdekoodianalyysityökalut, loogisten mallien purkajat, mittarit, debuggerit, testityökalut ja hyvät aikuisikään ehtineet vakaat kääntäjät. NASA on huomannut, että tietyt asiat toistuvat ohjelmointityössä ja ne on syytä kerrata säännöllisesti. NASA pyrkii listansa lisäksi rajoittamaan kaiken koodinsa yksinkertaisiin lyhyisiin funktioihin ja kontrollirakenteisiin, panostaa ennakoivaan virheenkorjaukseen ja he ovat myös varovaisia muistin käytössä. Koodin pitää myös aina kääntyä ennen kuin aloitetaan seuraavaa osaa ohjelmasta. Samoja ongelmia on havaittavissa jo ohjelmoinnin aloituskursseilla (ks. aloittelijoiden yleisimmät ohjelmointivirheet, AltadmriBrown 2015). Jo nämä perusasiat korjaamalla saavutetaan kohtuullinen laatutaso kaikkeen ohjelmointiin, joka on myös kriittisten järjestelmien rakentamisen tärkein perusta.

3 Kehittyminen kriittisen järjestelmän ohjelmoijaksi

Jotta voidaan miettiä miten kriittisen järjestelmän ohjelmointi tulee tehdä, on syytä ensin määritellä, mikä on kriittinen järjestelmä. On myös tärkeää muistaa, että kehittymispolkuja on monenlaisia kotiohjelmoijasta yliopisto-opiskelijaan. Harrastuneisuus kotioloissa auttaa oppimaan ohjelmointia ja siihen liittyviä teknologioita pienimuotoisesti. Työelämässä projektit ovat suurempia ja monimutkaisempia, jolloin on usein syytä kiinnittää huomiota suurempaan joukkoon tekijöitä. Jos ammattilainen luo reaaliaikajärjestelmän, on tärkeää ottaa huomioon reaaliaikaisuus, jolloin normaalin C-kielen ominaisuudet eivät riitä. Siksi eri kieliin on tehty laajennuksia jotka paremmin palvelevat näitä käyttötarkoituksia (mm. Broman 2018). Esimerkiksi Ada-ohjelmointikieli suunniteltiin alunperin reaaliaikajärjestelmille, mutta C:n, Javan ja muiden kielten suosio on nostanut ne Adan ohitse, jonka vuoksi niihin on pitänyt tehdä laajennuksia että ne toimivat paremmin. Kriittisissä järjestelmissä on siis omat lainalaisuutensa.

3.1 Mikä on kriittinen järjestelmä

Kriittiselle järjestelmälle on monia määritelmiä. Yksi määritelmä on, että kriittinen järjestelmä (*safety-critical system*) on järjestelmä jonka kaatuminen saattaa aiheuttaa ihmishenkien menetyksiä, suurta omaisuusvahinkoa tai vahinkoa ympäristölle (Knight, 2002). Tällaisia järjestelmiä löytyy sairaaloiden teho-osastoilta, lentomatkustamisesta, armeijan välineistöstä, ydinvoimaloista tai avaruusteknologiasta. Kriittiset järjestelmät kuitenkin yleistyvät kaikkialla ja niistä tulee jatkuvasti voimakkaampia. Esimerkiksi autoilussa olemme nähneet jo itseohjautuvia autoja, jolloin voidaan puhua kriittisestä järjestelmästä, koska järjestelmä itsessään voi aiheuttaa vaaratilanteen jollekin henkilölle. Lisäksi esineiden internet laajenee uusien laitteiden ansiosta. Pian lähes kaikki laitteet ovat verkottuneita moniälyisiä järjestelmiä, joilla on mahdollisuus tehdä sekä hyvää että pahaa.

Laitteiden laskevat kustannukset sekä laadun parantuminen antaa mahdollisuuksia

rakentaa kriittisiä järjestelmiä jo kotioloissa esimerkiksi ostamalla Raspberry Pi -tietokoneen ja rakentamalla kotiin sensorien avulla hälytysjärjestelmän. Teknologia kehittyy siis myös muilla osa-alueilla (verkot, sensorit, tietokonearkkitehtuuri).

Uusi monimutkaisempi maailma tuo mukanaan uusia monimutkaisempia ongelmia. Vuosituhannen alussa erääseen ydinvoimalaan rakennettiin ohjelmistoa, jonka piti sulkea ydinvoimala, jos eteen tulee jokin ongelma. Kun tällainen kriittinen järjestelmä saatiin luotua, se sisälsi lopulta 650 mikroprosessoria, 1200 piirilevyä ja 70 000 riviä koodia. Järjestelmä oli niin monimutkainen, että testatessa se läpäisi testeistä enää 48 prosenttia (Knight 2002). Monissa tapauksissa testiohjelmistot eivät enää edes tienneet, menivätkö ne läpi vai ei, koska kaikkia syy-seuraussuhteita ei saatu mallinnettua.

Englannista lanseerattiin projekti, jossa ambulanssien hälytysjärjestelmä vaihdettiin kerralla manuaalisesta digitaaliseen. Koska vaihtoa ei palasteltu pienempiin kokonaisuuksiin, seurasi käyttöönotto-ongelmia ja järjestelmä kaatui useiksi päiviksi (Knight 2002).

Ensimmäisillä Mars-lennoilla koettiin Mars Climate Orbiter -laitteen putoaminen ja tuhoutuminen, koska vääriä koodin osia käytettiin osassa maassa olevaa ohjelmistoa. Kävi ilmi, että insinöörit olettivat tiettyjen osioiden olevan käytössä, mutta näin ei käytännössä ollut (Knight 2002).

Vastaavia suuria järjestelmiä löytyy pian arkipäiväisemmistäkin pienlaitteista, jolloin voidaan sanoa, että kriittisten järjestelmien ohjelmointi on pian osa kaikkea ohjelmointia. Ohjelmakoodia, myös kriittisiä järjestelmiä, kannattaa tarkastella kokonaisuuksina, ei vain pieninä osina suurta ohjelmistoa (Knight, 2002). Ohjelmien tekeminen kestää koko ajan kauemmin, koska ne ovat hyvin laajoja. Tämän sovittaminen turvallisuuteen on suuri haaste tulevaisuudessa.

3.2 Kriittisten järjestelmien jaottelu

Yksityisellä puolella kriittisyys voi tarkoittaa myös sitä, että ohjelma noudattaa lakeja ja säädöksiä, turvaa liiketalouden jatkuvuuden ja säilyttää tärkeitä tietoja (Axelrod, 2011). Voidaan myös argumentoida, että talouden järjestelmät ovat jo kriittisiä. Vuoden 2008 pörssiromahdus toi tilanteen, jossa ihmishenkiä oli uhattuna. Kriittiset järjestelmät ovat usein yksityisiä tai teollisia prosessin kontrollijärjestelmiä, jotka rakennetaan ja testataan hyvin korkein standardein järjestelmävirheiden tai poikkeuksellisen toiminnan estämiseksi. Kriittisillä järjestelmillä on jo pitkä historia armeijan puolella, jossa vuosien mittaan on kehitelty erilaisia standardeja niiden luomisesta, ajamisesta ja ylläpidosta (esim *DO-178B, NIST'Special Publication 800-82 tai US Federal Information Security Management Act of 2002 (FISMA)*).

Yleisesti kriittiset järjestelmät voidaan jakaa kolmeen eri ryhmään: ei-kriittiset, tietoturvalliset ja kriittiset (*non-critical, security-critical, safety-critical*) (Axelrod, 2011). Ryhmittelyä tarvitaan suunnitteluvaiheessa, koska vaikkakin pankit, terveysjärjestelmät, energia ja liikenne omaavat tietoturvallisia ratkaisuja, ne eivät yllä samaan varmuuteen kuin asevoimissa tai lentämisessä on syytä. Insinööritkin voidaan jakaa eri ryhmiin: ne jotka keskittyvät tietoturvalliseen tietojenkäsittelyyn ja ne jotka keskittyvät turvallisuuskriittiseen prosessien kontrollointijärjestelmiin (Axelrod, 2011). Ongelmana nähdään, että nämä kaksi ryhmää eivät tunnu keskustelevan keskenään. John Barnesin sanoin, kommunikaatio on tärkeässä roolissa: "*Safety and security are intertwined through communication. Safety-the software must not harm the world, security-the world must not harm the software.*"Tämä jo asenteissa asti oleva eroavaisuus on huomioitava mm. siinä, että standardit tulisi kiinnittää jo suunnitteluvaiheessa, muuten niiden toteuttaminen on liiaksi kiinni yksittäisistä ihmisistä.

4 Kriittisen järjestelmän ohjelmointihaasteita

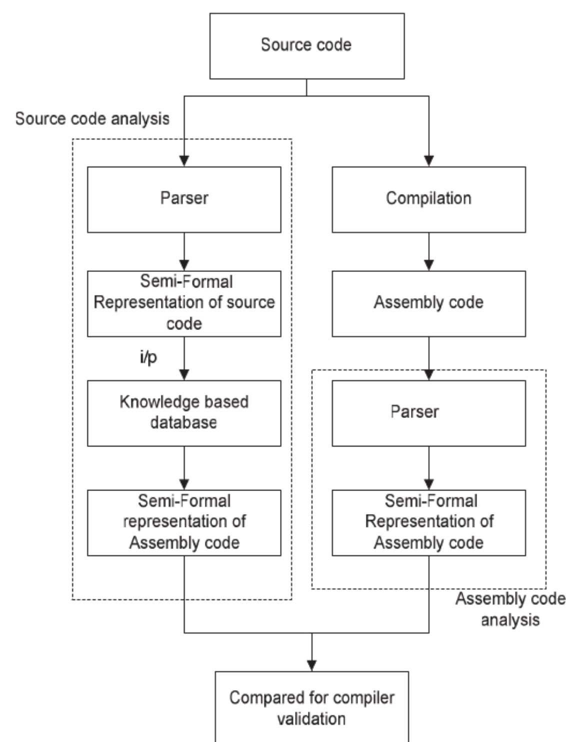
Laatu on prioriteetti kriittisissä järjestelmissä, joten testaus on ensiarvoisen tärkeää. Monimutkaisissa ohjelmistoissa testaaminen ei kuitenkaan aina tuo esiin kaikkia ongelmia. Jharko (2015) esittelee metodologian jolla laadusta voi pitää huolta. Lähtökohtaisesti koodin tulisi olla sellaista, että se täyttää vaatimukset, noudattaa suunnittelua ja seuraa ohjelmointikielen standardia. Ohjelman tulee tehdä vain sitä, miksi se on olemassa, väliaikaiset ja pysyvät resurssit pitää määritellä, ja kun virheitä löydetään, ne korjataan heti. Suunnittelussa ja vaatimuksissa turvallisuus tulisi olla aina määritelty jo ennen ohjelmointityön aloittamista, eikä ohjelmia tulisi korjata vain silloin, kun virhetilanne on jo päällä. Myös kommentoinnin tulee olla riittävää. Laatukriteerit ovat keskeinen määre, kun mietitään virheetöntä koodia. Se ei saisi olla ristiriidassa aika- tai rahamääreiden kanssa.

4.1 Kustannustehokkuus

Kriittisten ja kaikkien muidenkin ohjelmien tietoturvallisuus on tärkeämpää kuin koskaan. Vaikka maailmassa on paljon tietoa ja standardeja, meillä ei ole konsensusta siitä, miten kriittisiä ohjelmia tulisi ohjelmoida kustannustehokkaasti. Wong kumppaneineen (2011) analysoi viisi eri standardia sekä niiden kustannustehokkuuden: *FAA System Safety Handbook, US DoD MIL-STD-882D, UK MoD DEF-STAN 00-56, NASA-STD 8719.13b ja RTCA DO-178B*. Kyseessä ovat kriittisten järjestelmien keskeisiä standardeja, joiden mukaan ohjelmointia suoritetaan ympäri maailmaa. Tutkimuksen mukaan on täysin mahdollista kehittää ohjelmia näiden avulla ilman, että se maksaa merkittävästi enemmän. Usein kustannukset kuitenkin mielletään korkeiksi silloin, kun ne tulevat kerralla ja alussa. Kustannukset voivat kuitenkin pitkällä aikavälillä kasvaa paljon kovemmiksi johtuen huonosta kommunikoinnista, huonosta vaatimusmäärittelystä ja huonosta tai epätehokkaasta suunnittelusta (Wong 2011). Epätäsmällisyys projektin eri kohdissa tuntuu tuovan suurimmat kustannukset, ei turvallisuuden huomioonottaminen.

4.2 Pelkkä lähdekoodi ei riitä

Kriittiset järjestelmät ovat usein niin vaarallisia virhetilanteissa, että pelkkä lähdekoodin validoiminen ei riitä, vaan ohjelmat tulee testata objektikoodin tasolle asti (Avinash 2015). Ainoastaan ohjelmoijan kirjoittaman koodin tarkastelu on riittämätöntä, koska itse ohjelman suorittamiseen liittyy myös kääntäjä. Teknisenä esimerkkinä Stall-lentojärjestelmissä käytetään yhtenä vaihtoehtona Microtecin *cross-compiler* -kääntäjää jolla luodaan objektikoodia Motorola 68060-prosessorille. Kehittäjät käyttivät ohjelmointianalyysitekniikkaa kääntäjien validoimiseen, joka on heidän itse kehittämänsä työkalu. Sitä käytettäessä voidaan noudattaa RTCA DO 178B-säädöstöä (*Software Considerations in Airborne System and Equipment Certification*), joka on ilmailun alalla keskeinen standardi. Kääntäjä on hyvin kriittisessä roolissa, kun käännetään hyväksyttyä kriittistä lähdekoodia objektikoodiksi, joka sitten sulautetaan operaatiosysteemiin. Jos käännös ei ole sopiva, sulautettu koodi voi olla erilaista, joka johtaa järjestelmän kaatumiseen. On siis olennaista validoida koodin



Kuvio 1. Kääntäjäverifikointi (Avinash Brown 2015).

lisäksi kaikki kirjastot, työkalut ja erityisesti myös kääntäjät.

Kääntäjät myös usein lisäävät koodiin rivejä, jotka auttavat ohjelmaa toimimaan erilaisissa ympäristöissä sulavammin sekä käyttäjäystävällisesti. Kriittisiä järjestelmiä ohjelmoitaessa tällaiset lisärivit voivat johtaa riskialttiisiin tilanteisiin. Kohdeympäristö voi myös olla täysin erilainen kuin kehitysympäristö, joka tuo lisää haasteita. Tämän ongelman voi ratkaista luomalla kääntäjäkohtainen tietokanta (Avinash, 2015). Lähdekoodista etsitään käytetyt muuttujat, aritmeettiset operaatiot, loogiset operaatiot, suhteet ja kontrollirakenteet. Tästä muodostetaan analysoidulle lähdekoodille assemby-kielimuotoversio halutulle prosessorille. Tämän jälkeen luodaan vastaava assembly-koodi käyttämällä etukäteen tehtyä validoitua kääntäjäkohtaista tietokantaa. Näitä kahta koodia verrataan toisiinsa ja kerrotaan mitkä kohdat eroavat toisistaan. Lopputuloksena saadaan kääntäjäkoodia, joka on turvallista.

4.3 Puutteelliset työkalut

Yhdysvaltain puolustusministeriö on huomannut puutteita ohjelmointityön johdonmukaisuudessa, hyvien työkalujen varmistamisessa, metodologisuudessa ja testaamisessa (Axelrod, 2014). Suurimmat ongelmat löytyvät standardeista, työkaluista, teknologioista ja mittareista. Fokus tärkeimpiin asioihin tuntuu toisinaan puuttuuvan ja eritasoiset asiat arvotetaan samanarvoisiksi. Tavoitteena on tehdä riskejä vähentäviä hakuja (engl. *risk-reducing search*), joilla löydetään ne kohteet, joihin tulee aina keskittyä, kun suunnitellaan kriittisiä järjestelmiä. Jos halutaan muuttaa näitä kaikkia elementtejä, tarvitaan suunnanmuutos siinä, miten ohjelmia kehitetään, testataan tai määritellään. Yhdysvaltain puolustusministeriön mukaan kaikkien, jotka tekevät töitä kriittisten järjestelmien kanssa, tulisi vaihtaa tietoja keskenään paljon enemmän kuin tällä hetkellä. Ongelmana on, että yritykset ovat omillaan ja on paljon heistä itsestä kiinni miten hyvin ja mitä asioita tehdään turvallisuuden eteen. Rahaa tähän ei yleensä ole. Tarvittaisiin toisin sanoen uusia ohjelmointityökaluja, joilla tehdään ohjelmalliset hyökkäykset puolustuskyvyttömiksi. Myös uudet sertifikointi- ja testauslaboratoriot, avoimen lähdekoodin simulaatiot sekä musta laatikko-varmennukset laitteissa (engl. *black box*) parantaisivat ohjelmien ja laittei-

den laatua. Yhtenä esimerkkinä Malesian lento-onnettomuus 8.3.2014 saattoi johtua lentokoneen ohjausjärjestelmän sammumisesta siksi, että joku tai jokin katkaisi siitä virrat (Axelrod, 2014). Tätä inhimillistä mahdollisuutta ei oltu huomioitu suunnittelussa, jossa henkilökunnan jäsen katkaisee virrat tahallaan tai vahingossa, tai henkilökunta käyttää järjestelmää väärin. Vastaavista kokemuksista olisi Yhdysvaltain puolustusministeriön mukaan tärkeää vaihtaa kokemuksia, mieluiten etukäteen, jotta niistä voidaan oppia.

4.4 Konservatiiviset standardit eivät vastaa nykypäivää

Ilmailu on esimerkki toimialasta, jolla käytetään useita kriittisiä järjestelmiä. Ohjelmien turvallisuus varmistetaan kehitystyössä noudattamalla DO-178C standardia. Mielenkiintoisesti, siihen ei kuitenkaan sisälly ohjelmistojen turvallisuusnäkökulmaa (Bernsmed et al. 2018). Yksi ajankohtaisista kysymyksistä on, kuinka olemassaolevat ilmailun turvallisuusstandardit saadaan nykypäivän tietoturvaongelmaiseen maailmaan.

DO-178C Software Considerations in Airborne Systems and Equipment Certification on päädokumentti, jolla hyväksytään kaupallisia ohjelmapohjaisia lentojärjestelmiä. Toinen dokumentti, joka on myös kaikkien yritysten käytettävissä, on *BSIMM (The Building Security in Maturity Model)*, joka sisältää 113 havaittua tietoturvaongelmaa ilmailujärjestelmistä. Eroja DO-178C:n ja BSIMM:n välillä on mm. se, että DO-178C:ssä ei vaadita verifioinnin tekijältä lainkaan virallista osaamista, kuten tutkintotodistusta tai sertifikaattia. Ainoa ohjeistus on, että osaaminen on jonkun toimesta verifioitu. BSIMM taas suosittelee erityistä pätevyyttä mihin tahansa toimintoon. Muitakin eroja on, kuten se, että BSIMM haluaa koko henkilökunnan tietävän tärkeimmät ajankohdat ja *go/no-go* päätökset. DO-178C:ssä sanotaan ainoastaan yleisellä tasolla, että aktiviteetit on määritelty. DO-78C -tyyppiset perusstandardit lähtevät usein siitä, että ohjelmointivirheet ovat alun perinkin suunnitteluvirheitä, kun taas oikeassa maailmassa tietoturvaongelmia syntyy jatkuvasti teknologian kehittyessä. Perinteisissä kriittisissä järjestelmissä nopeita päivityksiä ei juuri tehdä, joka taas on nykyajan standardi kaikessa ohjelmointityössä. Kriittisten järjestelmien toimiala on siis

hyvin varovainen ja konservatiivinen. Yksi ohjelmoinnin haasteista onkin se, että sovitetaan turvallisiksi todetut vuosikymmeniä vanhat toimintamallit ja standardit tämän päivän muutoksia täynnä olevaan maailmaan.

4.5 Ohjelmointikielten riittävä määrittely

Kun tehdään kriittisiä ohjelmistoja, hyvin määritelty ohjelmointikieli on tärkeä osa ohjelmisto- ja järjestelmäprojektin valmistelua (Cullyer 1991). Sairaalalaitteiden mikroprosessorit, lentokoneiden navigointi, armeijan aseistus, ydinvoimaloiden monitorointi sekä hälytysjärjestelmät ovat osa korkeimman turvaluokan järjestelmiä (engl. *high integrity computing*). Yleensä tällaisissa järjestelmissä tietokone saa vaaratilanteessa joltain sensorilta syötteen, tekee laskelmia ja antaa toivotun tulosteen, eli toimenpiteen, jolla vaara mitätöidään. Järjestelmät ovat monikerroksisia ja sisältävät useita teknisiä spesifikaatioita sekä erityisen laadukkaita laitteita. Kun kieltä tai ohjelmointiympäristöä määritellään, on tällainen ympäristö huomioitava. Tärkeitä osa-alueita ovat onnettomuusanalysointi, matemaattiset spesifikaatiot, monikanavaimplementoinnin määrittely ja riippumaton verifikointi (Cullyer 1991). Esimerkiksi *UK Defence standard 00-55* sanoo, että kieltä pitää lisäksi pystyä käyttämään formaalin matematiikan ympäristössä. Kielen syntaksi ja semantiikka pitää olla tarkalleen tiedossa, jotta voidaan käyttää tekniikoita, kuten staattisen koodin analyysi (engl. *static code analysis*) sopivan laitteen kautta. Ei voida siis olla sen varassa, että löytyy henkilö, joka osaa harvinaisen kielen läpikotaisin, vaan luotetaan ennemmin logiikkaan. Kysymyksiä, joita tulee kysyä, kun uusi kieli valitaan, on mm. sisältääkö se rajoittamattomia hyppykäskyjä tai ylikirjoittamista, millaista semantiikkaa se sisältää, miten matematiikka mallinnetaan, millaista operationaalista aritmetiikkaa voidaan käyttää, miten tieto tyypitetään, miten muistia käsitellään, voidaanko ohjelma kääntää erillisesti, tai onko kieli hyvin ymmärretty ja tuettu (Cullyer 1991). Ada ja C ovat edelleen yleisesti käytössä juuri näistä syistä. Ihannetilanne on se, että on olemassa kielelle ominaiset ohjelmointikäytännöt, jotka on suunniteltu hyödyntämään tietyn nimenomaisen kielen ominaisuuksia.

4.6 Haastava fyysinen ympäristö

Kenties suurin eroavaisuus kriittisten järjestelmien ohjelmoinnilla verrattuna muuhun ohjelmointiin on se, että kohdelaite tai -henkilöt ovat usein vaarallisessa, jopa vihamielisessä, ympäristössä ja hengenvaarassa. Fahey (2016) kuvailee ohjelmistoarkkitehtuuria armeijan miehittämättömässä sukellusveneen etsintälaitteessa. Suunnittelussa pitää ottaa huomioon se, että laite on jatkuvassa vuorovaikutuksessa meriympäristön kanssa, jolloin ongelmia ovat suola, vesi, tuuli, aurinko, etäisyydet ja liikkeen vaikutus ihmiseen. Heidän suunnittelemansa autonomisen järjestelmän tulee siis olla niin laadukas, että se kestää kaiken mainitun sekä myös vihollisen, joka yrittää myös koko ajan häiritä tai tuhota kohdetta. Järjestelmän vaatimuksia voivat olla esimerkiksi tekoäly, näyttö, arvojen analysointimallit, kestävä fysiikka, piiloutumismallit, ylläpito, signaalien lukeminen, navigointi ja kommunikointi. Lisäksi tulee huomioida itse laitteiden ohjelmointi esimerkiksi HDL-kielellä. Tästä syystä armeija pitäytyy usein vedenpitävissä standardeissa ja säännöissä, sillä se ei halua lisäongelmia ohjelmointivirheistä. Tämä on ristiriidassa tämän päivän nopean kehityssyklin kanssa, joka jälleen tuo oman haasteensa kriittisten järjestelmien ohjelmointiin. Useita ratkaisuja on silti olemassa.

5 Kohti järjestelmien ammattimaista ohjelmointia

Ammattimainen koodi on selkeää, luettavaa ja turvallista. Koodia saattaa lukea useat eri henkilöt ydinvoimalassa, teho-osastolla, meren pohjassa, avaruudessa tai lentokoneessa. Nämä henkilöt eivät tunne ohjelman tekijää ja heillä ei ole välttämättä käytössään viimeisimpiä työkaluja tai tehokkainta konetta. Tästä syystä ohjelmien on järkevää olla sellaisia, jotka tekevät vain sen ydintehtävän, mikä niille on luotu.

Kriittisten järjestelmien ohjelmoinnissa on olemassa monia keinoja, joilla turvallisuutta voi lisätä. Jos esimerkiksi ohjelmistoja on verkossa, mietitään mitkä osiot ovat suoraan yhteydessä maailmaan ja mitkä osat eriytetään (engl. *air caps*). Tällöin ohjelmiin ei ole suoraa pääsyä internetin kautta. Ohjelmointikielistä tulisi riisua pois turhat hienostelevat ominaisuudet, jolloin koodi on luettavampaa. Kriittiset osiot voidaan tehdä supistetulla mutta toimintavarmemmalla kielellä (ns. *system hardening*). Yleissääntö on se, että ne kielet, joissa pakottavat kurinalaisuuteen ja itse asioiden kirjoittamiseen ja ymmärtämiseen, tuovat parhaat tulokset. Näillä kielillä ohjelmoiminen tosin on usein työläämpää ja vaatii enemmän ammattitaitoa (Axelrod, 2011). Myös käyttöjärjestelmistä voi olla käytössä riisutummat mallit, kuten Linuxin eri jakeluversiot. Samoin laitearkkitehtuurit sisältävät ennaltaehkäiseviä työkaluja ja niissäkin kaikkein kriittisimmät toiminnot eristetään muista. Laitteistojen kokoonpano on tarkkaan määritelty. Koodauskäytännöissä käytetään parityöskentelyä (*peer review*), verifiointia ja validointia. Ammattimaisuus on kaiken kaikkiaan ohjelman käyttäjien ja ympäristön huomioimista, sekä laadukkaan ydinohjelman rakentamista.

5.1 Ohjelmointikielen rajattu käyttö

Kuten edellä on mainittu, kriittisissä järjestelmissä on yleisesti suosittua käyttää kieliä joita on käytetty paljon, joilla on runsaasti resursseja ja kirjastoja, ja jotka ovat joustavia, tehokkaita ja kompakteja (HaighLandwehr 2015). Näissä kielissä voi kuitenkin tehdä virheitä, joita voi olla vaikea löytää. Tämä antaa sekä hyökkääjille että

virheille enemmän mahdollisuuksia. Kielissä, kuten C ja C++, ongelmia syntyy esimerkiksi muistin käytössä, alustamisessa ja sen vapauttamisessa, tai *buffer overflow / null pointer* -viittauksissa. Myös pointtereiden käyttö on myös vaarallista kriittisissä järjestelmissä. Nykyään on olemassa myös sopivia kieliä, jotka estävät näitä muistivirheitä. Käyttäjiä voi myös kouluttaa. Yksi parhaista keinoista estää virheellistä koodia on kuitenkin käyttää kielen versiota jossa voi koodata vain rajattua aluetta (esim *MISRA C tai SPARK Ada*)(HaighLandwehr 2015). Minimivaatimuksena on käyttää aina turvallisen ohjelmoinnin standardeja, joita on olemassa kaikille kielille.

Edellä mainittiin kääntäjien tietoturvallinen käyttö. Tämän lisäksi on myös kääntämistekniikoita, jotka painottavat turvallisuutta, kuten *SAFECode, WIT, Baggy Bounds Checking ja SoftBound*. Käytössä on myös automaattisia ohjelmia, jotka tutkivat, löytyykö lähdekoodista kohdelaitteiston ja ohjelmaympäristön ympäristön kriittiset osiot. On siis monia vaihtoehtoja toimia "rajatusti"myös epäturvallisissa ympäristöissä tai testata ohjelmiston eri olosuhteita (HaighLandwerh 2015). Yleinen malli ilmailualalla, sekä rautateiden ja autoalan standardeissa on luoda erilaisia haastavia olosuhteita, jonka jälkeen katsotaan, miten ohjelmat niihin reagoivat. Tämä ns. *use case analysis* nostaa esiin koodista monia sen puutteita. Yksi yleisimpiä toimenpiteitä usein on se, että jos jotain osaa ohjelmasta ei käytetä, se poistetaan. Tämä jo aiemmin mainittu turhan hienosteleva koodi tulee siis lopulta poistettua ohjelmasta.

On myös suositeltavaa käyttää laitteita ja laitteiden ohjelmistoja (*firmware*), joissa on digitaalinen valmistajan allekirjoitus, jonka saa standardia vastaan. Ns. *whitelisting* varmistaa, että käytetään vain luotettuja ohjelmia ja valmistajia. Haasteena tuntuu olevan, että ohjelmien tekijöillä ei ole insentiiviä käyttää näitä listoja.

5.2 Apua matematiikasta

Myös matematiikka voi tarjota apukeinon ammattilaisohjelmoijalle, mikä sinänsä ei ole yllättävää. Koska jatkossa kriittisiä järjestelmiä on kaikkialla, olisi loogista, että ohjelmoijat kokisivat ohjelmointikielten lisäksi mielenkiintoa matematiikkaa kohtaan. Yksi esimerkki matematiikan soveltamisesta ohjelmoinnissa on se, että ensin

tehdään toimiva minimaalinen perustilanne, perusohjelma. Vasta sen jälkeen edetään kehittyneempään versioon. Toisena ajatusmallina voisi olla tehdä tarkistukset aina sellaista tilannetta vastaan, josta olet varma että se toimii, joka on luotettava tai jota vasten on aiemminkin tarkistettu. Hyvä strategia on myös palastella, tuottaa ja testata pienissä paloissa kerrallaan. Logiikka tai matemaattinen todistaminen ovat helppoja esimerkkiä vuosisatoja vanhoista tekniikoista, joista on hyötyä myös ohjelmoijalle.

Ohjelmistoprojektit ovat usein suuria, satatuhatrivisiä projekteja. Jos kerralla tarkistetaan tuhansia rivejä koodia, erilaisten mahdollisuuksien ketjut ovat eksponentiaalisia, jolloin virhetilanteen sattuessa tarkistaminen on usein liian suppeaa ja myöhäistä. Edelleen tehdään myös ohjelmia ilman alun määrittelyjä. Testaamista pyritään tekemään tehokkaasti, jolloin ei aina tiedetä miten koodi, ohjelmisto tai laitteet toimivat pitkällä aikavälillä. On myös olemassa psykologinen ongelma, jossa virhe löytyy ja se pitäisi korjata, mutta etenkin pienten virheiden korjaamista vältellään, koska se saattaa aiheuttaa uusien virheiden ketjureaktion. yrityksen johto saattaa miettiä kannattaako testata koko koodia, koska ohjelmisto menettää rahallisen arvonsa ja tärkeytensä siinä ajassa.

On runsaasti syitä testata ohjelmistoja automaattisesti ja luotettavasti. Matematiikka tarjoaa runsaasti malleja joita hyödyntää. Keksijöitä on vuosisatojen ajalta: Turing, von Neuman, Igarashi, Floyd, Jones, Gries (Hoare 2015). Ammattiohjelmoija ei hyväksy virheitä, vaan hyödyntää kaikki luotettavat työkalut ja pitkän tieteen historian lähtiessään rohkeasti kohti virheetöntä koodia. Valmiita ohjelmakirjastoja on tähänkin jo olemassa.

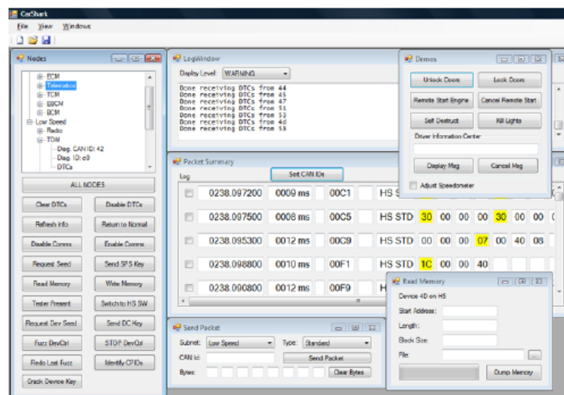
5.3 Myös laitteet tulee ohjelmoida

Yhtenä alueena nykypäivän kriittisissä järjestelmissä on myös se, että kyse ei ole vain ohjelmista. On yhä kasvava määrä räätälöityä laitteistoja, tietomassoja, kontrollijärjestelmiä ja pienlaitteita jotka ovat myös ohjelmoitavia. Näiden koodaus tehdään sekä normaaleilla ohjelmointikielillä että laitteistojen kuvauskielillä (*HDL, Ve-*

rilog, Assembly) jolla luodaan mm. laitteiden osille logiikka. Tällöin käytetään ohjelmia, joilla simuloidaan laitteiden logiikkaa, syntetisoidaan piirejä, sijoitetaan ja reititetään elektronisia elementtejä, sähköä sekä verkotuksia (mm. *programmable logic devices (PLD), field-programmable gate arrays (FPGA), application-specific integrated circuits (ASIC)*). Järjestelmissä yhdistyvät sekä sovellus- ja sokettiohjelmointi, laitteiden ohjelmointi että verkkojen ja sensoreiden konfigurointi. Tällöin on kiinnitettävä huomiota paljon laajemmassa kontekstissa turvallisuuteen ja virheettömyyteen. Tällä hetkellä piirien ohjelmointia ei vielä pidetä kaikkialla ohjelmointina, koska sitä tekevät lähinnä koneinsinöörit (vaikkakin heidän käyttämänsä laitteiden kuvauskielet ovat syntaksiltaan ja semantiikaltaan ohjelmointikieliä). Onneksi asia on kuitenkin muuttumassa. Kriittisissä järjestelmissä ei voida erotella näitä kahta. On vain yksi kokonaisuus, jonka tulee täyttää standardit. Molempien, sekä ohjelmoijien että insinöörien, tulee ymmärtää toistensa työtä.

5.4 Seppä ja kisälli - Opi tekemällä

Ohjelmointi on käsityötä ja sitä voi oppia tehokkaasti toisilta ohjelmoijilta. Vihavaisen ja kumppaneiden lanseeraama *Exreme Apprenticeship*(2011) on tekemällä oppimista, parhaista käytännöistä, jo tekeviltä ihmisiltä. He esittelevät selkeitä tuloksia, joissa oppimista on saavutettu. Heidän tutkimissaan ohjelmointikursseissa läpipää-



Kuvio 2. Auto on esimerkki yhdistelmästä ohjelmia ja laitteiden ohjelmointia. Kuvassa autonkaappausohjelma (Koscher et. al 2010).

syprosentit kasvoivat peruskurssilla 60:stä 70:een prosenttia ja jatkokurssilla 60:sta 80:aan prosenttiin, kun Seppä ja kisälli -ohjelmoinnin opetteleminen otettiin käyttöön.

Tekemällä oppiminen yhdessä hyvien ohjelmointikäytäntöjen kanssa voi olla ratkaisu sekä kriittisten järjestelmien että kaiken muunkin ohjelmoinnin haasteisiin. Kun yhdistetään vankka teoriapohja käytännön harjoitteluun, tulokset näyttävät olevan hyviä. Vihavaisen ym. tekemän tutkimuksen aikana kerättiin nimetöntä palautetta opiskelijoilta miten he kokivat Seppä ja kisälli -tyyppisen opettamisen tapana hallita ohjelmoinnin perusteita. Tulokset osoittivat selvästi, että opiskelijat arvostivat tämän tyyppistä oppimista alustana. Vakuuttavimmat tulokset tulivat opiskelijoilta, jotka eivät aiemmin läpäisseet ohjelmointikurssia, mutta tällä tavalla kurssit saatiin läpäistyä.

Yhdessä tekeminen voi olla auttamista silloin, kun on avun tarve ja henkisesti olla läsnä tuomassa itseluottamusta ohjelmointitehtäviin. Kun seppä saa kisällin puhumaan virheistään ilman tuomitsemista, lopputulos voi parantua (Vihavainen et al. 2011). Kenties *Extreme Programming* -tyyppinen työskentely voi olla avuksi myös kriittisten järjestelmien rakentamisessa ja ohjelmoinnissa, jolloin minimoidaan sekä virheitä että luodaan turvallinen kehitysympäristö virheherkille järjestelmille.

6 Yhteenveto

Tämä kirjallisuuskatsaus antaa yleiskuvan kriittisten järjestelmien ohjelmointiin liittyvistä haasteista ja ratkaisuista. Tutkielma esittelee ohjelmointitaitoa yleisesti sekä siihen liittyviä yleisimpiä virheitä sekä antaa ohjeita parempaan koodiin kaikkialla. Kriittiset järjestelmät määritellään terminä ja niihin liittyvää ohjelmointia käsitellään haasteineen ja ratkaisuineen. Monet haasteet sekä ratkaisut ovat yhteneväisiä sekä kriittisten järjestelmien ohjelmoinnissa että missä tahansa ohjelmointityössä, kuten huolimattomuus ja kiire, rahan ja määrän merkitys verrattuna laatuun, kaupallisten järjestelmien yksiulottuvuus ja standardien käsittelminen. Kriittisille järjestelmille omanlaisena ongelmana nostetaan esiin erikseen haastava fyysinen ympäristö.

Tutkielma käy läpi kriittisten järjestelmien ohjelmoinnin haasteita ja ratkaisuja ja antaa työkaluja kohti ammattimaista ohjelmointia sekä kriittisissä järjestelmissä että ohjelmoinnissa yleisesti miettimällä ohjelmointikielten rajatumpaa käyttöä, matematiikkaa apukeinona ja turvattomien kielten järkevää käyttöä. Lisäksi mietitään uusia tapoja harjoitella ohjelmointia seppä ja kisälli -tyyppisen ajattelun kautta.

Lisää tutkimusta on syytä tehdä siitä, miten erilaiset ympäristöt ja käyttäjäryhmät vaikuttavat ohjelmoinnissa vallalla oleviin standardeihin. Tulevaisuudessa kriittisiksi järjestelmiksi voitaneen luokitella lähes kaikki ohjelmointityö, joka tapahtuu laitteiden, palveluiden tai ohjelmien parissa jotka ovat verkottuneet keskenään globaalisti. Tämä tuo eteen tilanteen, jossa yhä uusia ihmisryhmiä ja ympäristöjä liittyy mukaan tietotekniseen perheeseen, joka tuo uusia tietoturvahaasteita mutta myös näkökulmia ohjelmointityöhön. Kun yhä uudet toimijat liittyvät mukaan tekemään kriittisiä järjestelmiä, niiden ohjelmoinnista tulee yhä suurempi haaste. Kriittiset järjestelmät paisuvat suuriksi laitteiden ja ohjelmoien verkoiksi, joissa tietoturvalliset virheitä estävät standardit tulevat ensiarvoisen tärkeiksi työkaluiksi.

Kirjallisuutta

Altadmri, A. & Brown, N.C.C. 2015. *37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data.*. Proceedings of the 46th ACM Technical Symposium on Computer Science Education. CSE Computer Science Education . ACM, New York, USA, s. 522–527. ISBN 978-1-4503-2966-8. doi:10.1145/2676723.2677258.

Avinash, A.K., Nanda, M. & Jayanthi, J. 2015. *Semi-formal approach for validating compiler for safety critical software in airborne systems.*. 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom) s. 854–858. IEEE.

Axelrod, C.W. 2011. *Applying lessons from safety-critical systems to security-critical software.*. 2011 IEEE Long Island Systems, Applications and Technology Conference, Farmingdale, NY, 2011, s. 1–6. doi: 10.1109/LISAT.2011.5784222.

Axelrod, C.W. 2014. *Reducing software assurance risks for security-critical and safety-critical systems.*. IEEE Long Island Systems, Applications and Technology (LISAT) Conference 2014, Farmingdale, NY, 2014, s. 1–6. doi: 10.1109/LISAT.2014.6845212.

Bernsmed, K., Jaatun, M.G., Meland, P.H. 2018. *Safety Critical Software and Security-How Low Can You Go?*. 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC) (pp. 1–6). IEEE.

Cullyer, W.J., Goodenough, S.J. & Wichmann, B.A. 1991. *The choice of computer languages for use in safety-critical systems.*. Software Engineering Journal, 1991 Volume 6(2), s.51–58. IEEE.

Delic, E., Loser, K., Hayek, A. & Borcsok, J. 2015. *Platform independent safety-critical operating system.*. 2015 International Conference on Information and Digital Technologies, Zilina, 2015, s. 68–71. doi: 10.1109/DT.2015.7222952.

Fahey Jr, S. 2016. *Software architecture for anti-submarine warfare unmanned surface vehicles.*. Naval Postgraduate School, Monterey, United States.

Haigh, T. & Landwehr, C. 2015. *Building code for medical device software security.*. Tekninen workshop-dokumentti. IEEE Cybersecurity.

Hoare, C.A.R. 2015. *Maths adds safety to computer programs.*. Alkup. Syyskuun 18.

1986 artikkeli, New Scientist s.53–56.

Holzmann, G. 2006. *The Power of 10: Rules for Developing Safety-Critical Code.*. IEEE Computer. Volume 39. s. 95–97. doi: 10.1109/MC.2006.212.

IEEE. 1994. *IEEE Standard for Software Safety Plans.*. 1994 IEEE Std 1228-1994, tekninen ISO-dokumentti. doi: 10.1109/IEEESTD.1994.122165

Jharko, E.P. 2015. *The methodology of software quality assurance for safety-critical systems.*. 2015 International Siberian Conference on Control and Communications (SIBCON), Omsk, 2015, s. 1–5. doi: 10.1109/SIBCON.2015.7147057.

Knight, J.C. 2002. *Safety critical systems: challenges and directions.*. Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, Orlando, FL, USA, 2002, pp. 547–550.

Kornecki, A., Butka, B. & Zalewski, J. 2009. *Software Tools for Safety-Critical Systems According to DO-254.*. Computer. Volume 41. s. 112–115. doi: 10.1109/MC.2008.503.

Koscher, K., Czeskis, A., Roesner, F., Patel, S. & Kohno, T. 2010. *Experimental Security Analysis of a Modern Automobile.*. 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, 2010, s. 447–462. doi: 10.1109/SP.2010.34.

Natarajan, S. & Broman, D. 2018. *Timed C: An Extension to the C Programming Language for Real-Time Systems.*. 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) s. 227-239. IEEE.

Pattabiraman, K., Grover, V. & Zorn, B. 2008. *Samurai: Protecting critical data in unsafe languages.*. Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008 Volume 42. s. 219–232. doi: 10.1145/1352592.1352616.

Pinheiro, V.H.C. & Schirru, R. 2019. *Genetic programming applied to the identification of accidents of a PWR nuclear power plant*. Annuals of Nuclear Energy, Volume 124, 2019, s 335–341, ISSN 0306-4549. https://doi.org/10.1016/j.anucene.2018.09.039.

Shi, J.Y.. 2019. *On the resilience of mission critical applications.*. 2015 Resilience Week (RWS), Philadelphia, PA, 2015, s. 1–3. doi: 10.1109/RWEEK.2015.7287445.

Turner, S. 2014. *Security vulnerabilities of the top ten programming languages: C, Java, Cplusplus, Objective-C, Csharp, PHP, Visual Basic, Python, Perl, and Ruby.*. 2014 Journal of Technology Research, Volume 5, s.1.

Vihavainen, A., Paksula, M., Luukkainen, M. & Kurhila, J. 2011. *Extreme apprenticeship method: Key practices and upward scalability.*. Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, 2011, s. 273–277. doi: 10.1145/1999747.1999824.

Wong, W., Demel, A., Debroy, V. & Siok, M. 2011. *Safe Software: Does It Cost More to Develop?*. Proceedings - 2011 5th International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011. doi: 10.1109/SSIRI.2011.28.