Tampereen yliopisto

Petri Sydänmaa

TESTIPENKKIEN KEHITTÄMISEN AUTOMATISOINTI FUNKTIONAALISESSA VARMENNUKSESSA

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Sakari Lahti
Tarkastaja: Arto Oinonen
Toukokuu 2022

TIIVISTELMÄ

Petri Sydänmaa: Testipenkkien kehittämisen automatisointi funktionaalisessa varmentamisessa
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan tutkinto-ohjelma
Toukokuu 2022

Varmennus on laitteistokehityksen yksi eniten työtä vaativista vaiheista. Tässä työssä tutkitaan, miten varmennukseen vaadittavaa työtä voidaan vähentää automatisoimalla varmentamiseen käytettävien testipenkkien kehitystä käytettäessä universaalia varmennusmenetelmää (engl. Universal Verification Methodology, UVM). Työn tavoite on selvittää minkälaisia hyötyjä testipenkkien kehittämisen automatisointi tuottaa varmennukseen.

Työ jakaantuu kahteen osaan. Teoriaosa esittelee varmennukseen liittyvän tehokkuusongelman, universaalin varmennusmenetelmän ja aikaisemman tutkimuksen testipenkkien kehittämisen automatisoinnista. Työssä havaitaan, että aiempi tutkimus keskittyy kehittämään työkaluja testipenkkien kehittämisen automatisointiin perinteisillä laitteistokuvauskielillä, kun taas UVM-testipenkkien kehittämisen automatisointiin tähtäävää tutkimusta on niukalti tai ei lainkaan. Täten UVM-testipenkkien kehittämisen automatisointiin liittyvät järjestelmät ovat yksinomaan kaupallisia työkaluja. Tutkimusosiossa toteutetaan UVM-testipenkki hyödyntäen Siemensin UVM Framework -työkalua. Osiossa esitellään työkalun vaatima kuvaus testipenkin rakenteesta sekä luotuun testipenkkiin vaadittavat lisäykset varmennuksen toteuttamiseksi.

Tutkimus osoittaa, että hyödyntämällä automatisoitua testipenkin kehitystä, voidaan varmennukseen vaadittavaa työmäärää vähentää, sillä käsin kirjoitettavan lähdekoodin määrä vähenee huomattavasti. Toisaalta nähdään, että työkalun tuottama ajallinen hyöty testipenkin kehityksessä ei ole ilmiselvä ja työkalu voi lisätä testipenkkiin ylimääräistä monimutkaisuutta.

Avainsanat: Systemverilog, UVM, automatisointi, testipenkki, varmennus

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

DUT	engl. design under test, varmennettava laitteistokuvaus
DUV	engl. design under test, varmennettava laitteistokuvaus
Systemverilog	laitteistokuvaus- ja varmennuskieli
TLM	engl. transaction level modelling, kommunikaatiomallinnuksen abstraktiotaso
UVM	engl. Universal Verification Methodology, Universaali varmennusmenetelmä
UVMF	engl. UVM Framework, työkalu testipenkkien kehityksen automatisointiin
Verilog	laitteistokuvauskieli
VHDL	engl. Very High Speed Integrated Circuit Description Language, laitteistokuvauskieli
YAML	engl. YAML Ain’t Markup Language, merkintäkieli

1. JOHDANTO

Viime vuosikymmeninä digitaalipiirien transistorimäärät ovat kasvaneet huomattavasti. Tämä johtuu puolijohdeteknologian nopeasta kehittymisestä, jota kuvaamaan on yleistynyt Mooren laki, jonka mukaan transistorien määrä digitaalipiirillä kaksinkertaistuu kahden vuoden välein. Suurempi transistorien määrä tarkoittaa, että yksittäiselle piirille pystytään mahduttamaan enemmän digitaalilogiikkaa, mikä on johtanut piirien monimutkaisuuden ja logiikan määrän voimakkaaseen kasvuun.

Monimutkaisuuden ja logiikan määrän kasvaessa digitaalipiirien suunnittelu kuitenkin vaikeutuu vaatien paljon työtä, aikaa ja rahaa. Erityisesti piirien monimutkaisuus vaikeuttaa piirien toimivuuden osoittamista, varmennusta. Monimutkaisten piirien varmennus on haastavaa, joten sen nopeuttamiseksi ja helpottamiseksi on kehitetty erilaisia menetelmiä ja työkaluja. [1]

Näistä yleisin on universaali varmennusmenetelmä, joka on menetelmä ja ohjelmointirajapinta testipenkkien kehittämiseksi. Testipenkki on ohjelma, joka simuloidaan digitaalilogiikkaa kuvaavan lähdekoodin, laitteistokuvauksen, kanssa. Testipenkki suorittaa varmennuksen syöttämällä simuloidulle laitteistokuvaukselle testisyötettä ja tarkkailemalla laitteistokuvauksen vastausta [2]. Tästä käytetään termiä funktionaalinen varmennus. Tässä työssä termillä varmennus tarkoitetaan aina funktionaalista varmennusta.

Tässä työssä tutkitaan, miten testipenkkien kehityksen automatisointia voidaan hyödyntää laitteistokuvausten varmennuksen työmäärän vähentämiseksi. Erityisesti työ keskittyy kehityksen automatisoinnin tutkimiseen käytettäessä universaalia varmennusmenetelmää. Sitä hyödynnettäessä kirjoitettavan lähdekoodin määrä on suuri, joten lähdekoodin luominen työkalun avulla on varteenotettava vaihtoehto varmennukseen vaadittavan työmäärän ja ajan vähentämiseksi. Testipenkin kehitykseen vaadittavaa työmäärää arvioidaan tässä työssä mittaamalla käsin kirjoitettavan lähdekoodin määrää.

Työ on järjestelty seuraavalla tavalla. Luku 2 esittelee ongelman tutkimuskysymyksen taustalla, universaalin varmennusmenetelmän ja aikaisempaa aiheesta tehtyä tutkimusta. Luvussa 3 tutkitaan testipenkin automatisoidun kehityksen hyötyjä kehittämällä testipenkki automatisointia hyödyntäen. Luvussa 4 esitellään luvun 3 testipenkin kehityksestä saatavat tulokset. Luku 5 sisältää yhteenvedon työstä.

2. VARMENNUS DIGITAALIELEKTRONIIKASSA

2.1 Varmennuksen tehokkuusongelma

1990-luvun loppupuolella piirien transistorimäärät kaksinkertaistuivat kahden vuoden välein Mooren lain mukaan, kun taas laitteistosuunnittelun tehokkuus kasvoi vain noin 20 %:n vuositahdilla. Tällöin Sematech ennusti, että tarve uusille laitteistoinsinööreille kasvaisi huomattavasti, jotta suunnittelutehokkuus pysyisi valmistusmenetelmien kehityksen tasolla [1]. Tämä aiheutti huolen, että mikropiirien suunnittelussa ei kyettäisi hyödyntämään uusimman puolijohdeteknologian mahdollistamia transistorimääriä, koska piirien suunnittelu olisi liian kallista ja hidasta. [3]

Tämä suunnittelun tehokkuusongelma ei kuitenkaan realisoitunut, mihin vaikutti varsinkin uudelleenkäytön lisääntyminen laitteistokehityksessä [1]. Alan yritykset alkoivat integroida enemmän kolmannen osapuolen moduuleja omiin järjestelmiinsä sekä lisäsivät omien komponenttiensa uudelleenkäyttöä uusissa laitteistoprojekteissa. [4]

Kehitysaskeleet sekä piirien suunnittelussa että valmistuksessa ovat johtaneet laitteistojen monimutkaisuuden nopeaan kasvuun. Monimutkaisuuden kasvaessa varmennukseen vaadittava työmäärä kasvaa huomattavasti nopeammin kuin suunnitteluun vaadittava työmäärä. Tämä ilmiö on aiheuttanut tehokkuusongelman suunnittelun ja varmennuksen välille tehden varmennuksesta pullonkaulan monissa laitteistoprojekteissa. [5]

Digitaalielektroniikan varmennuksella tarkoitetaan suunnitellun laitteistokuvauksen oikean toiminnan osoittamista. Tyypillisesti suunniteltavan laitteistokuvauksen toiminta määritellään komponentin määrittelydokumentissa. Tällöin laitteistokuvauksen oikean toiminnan osoittamiseen voidaan käyttää testipenkkiä, joka syöttää testisyötettä laitteistokuvaukselle ja lukee sen reaktiota syötteeseen verraten sitä määrittelydokumentin mukaiseen toimintaan. Koska laitteiston valmistaminen on kallista ja hidasta, varmennusta ei ole kannattavaa suorittaa fyysiselle piirille vaan yleisimpänä työkaluna käytetään laitteistosimulaattoria. Tällöin testipenkillä tarkoitetaan simulaattorissa suoritettavaa ohjelmaa, joka simuloidaan laitteistokuvauksen kanssa. [2]

Piirisuunnittelun tehokkuutta kehitti erityisesti uudelleenkäyttö, mutta samat menetelmät eivät todennäköisesti ratkaise varmennuksen tehokkuusongelmaa, sillä uudelleenkäyttö on varmennuksessa jo yleistynyt [1]. Uudelleenkäytön lisäksi varmennuksen tehostamiseksi tarvitaan testipenkkien kehittyneitä ominaisuuksia. Näitä ovat esimerkiksi satunnaistaminen ja funktionaalinen kattavuus. Satunnaistamisella tarkoitetaan

testattavan laitteistokuvauksen ja testipenkin konfiguraation sekä testisyötteen satunnaistamista. Kohdennettujen testien kirjoittaminen korkean kattavuuden saavuttamiseksi on työlästä, mutta satunnaistamalla kattavuutta saadaan kasvatettua suorittamalla satunnaistettuja simulaatioita tarpeeksi monta kertaa. Funktionaalisella kattavuudella tarkoitetaan laitteiston varmennettujen ominaisuuksien seuraamista koodikattavuuden sijaan. Tällä tavoin voidaan myös satunnaistettua simulaatiota jatkaa, kunnes funktionaalinen kattavuus on saavuttanut tavoitellun rajan.

Vaikka uudet menetelmät ja työkalut sekä uudelleenkäyttö ovat yleistyneet varmennuksessa, varmennusinsinöörien määrä sekä tarve on silti kasvanut 2010-luvulla. Useimmissa projekteissa varmennusinsinöörien määrä on ohittanut laitteistoinsinöörien määrän varmennukseen vaaditun ajan ylittäessä laitteiston suunnitteluun vaaditun ajan [6]. Piirien kasvava monimutkaisuus sekä lyhyempi aika markkinoille asettavat edelleen haasteita varmennuksen tehostamiselle. [1]

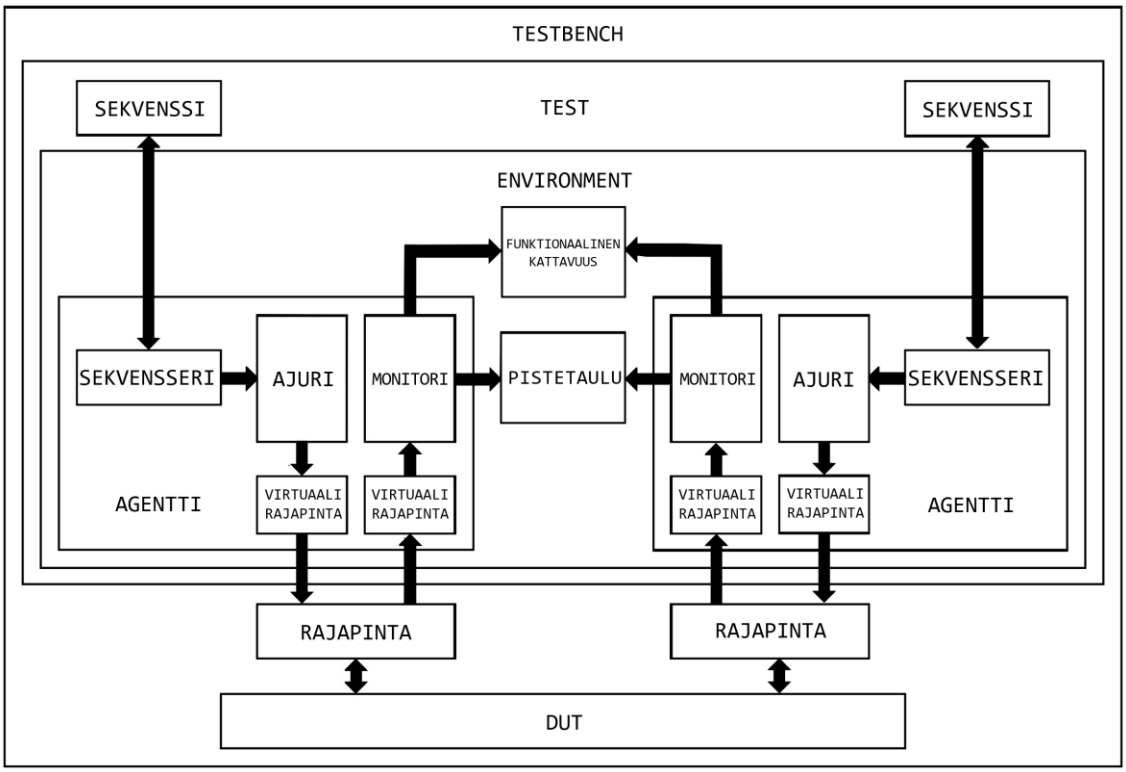
Varmennuksen tehokkuuden parantamiseksi on kehitetty uusia varmennusmenetelmiä. Perinteiset laitteistokuvauskielillä kirjoitetut testipenkit ovat liian joustamattomia ja tehottomia. Ne kärsivät kehittyneempien menetelmien heikosta tuesta sekä huonosta uudelleenkäytettävyydestä [7]. Eri työkaluvalmistajat kehittivät erillisiä ohjelmointikieliä ja metodeja varmennukseen. Nämä toivat paremman tuen uudelleenkäytölle ja kehittyneille ominaisuuksille, mutta eivät toimineet ristiin toistensa kanssa eivätkä kaikki simulaattorit tukeneet kaikkia menetelmiä ja kieliä [2]. Lisäksi erilliset varmennusmenetelmät vaativat koulutusta ja erilaisten työtapojen omaksumista ja käyttöä. Yksi kieli ja metodologia, joita kaikki simulaattorit ja työkaluvalmistajat tukisivat, olisi joustavampi vaihtoehto varmennuksen näkökulmasta.

2.2 Universaali varmennusmenetelmä

Vuonna 2011 Accellera julkaisi Universaalin varmennusmenetelmän (UVM). Accellera ei ole työkaluvalmistaja vaan standardiorganisaatio, joten sen kehittämä metodologia on käyttökelpoinen ja avoin kaikille työkaluvalmistajille [8]. UVM perustuu Verilog-laitteistokuvauskielen laajennokseen Systemverilogiin. Varmennuksen näkökulmasta Systemverilogin tärkein ominaisuus Verilogiin verrattuna on tuki olio-ohjelmoinnille. UVM määrittelee ohjelmointirajapinnan sekä useita luokkia testipenkkien kehittämiseen olio-ohjelmointia hyödyntäen. UVM ei ole työkalukohtainen metodi vaan standardi varmennusmenetelmälle, jota työkaluvalmistajat tukevat. [2]

UVM:n ja Systemverilogin olio-ohjelmointiin perustuva metodologia helpottaa testipenkin jakoa pienempiin, uudelleenkäytettäviin osiin. Lisäksi UVM erottelee erillisiin

kokonaisuuksiin testipenkin ympäristön, testisekvenssit sekä testit. Kuvassa 1 esitetään tyypillinen UVM-testipenkin arkkitehtuuri.



Kuva 1. *Tyypillinen UVM-testipenkin arkkitehtuuri, muokattu lähteestä [9]*

Testipenkin ympäristö sisältää agentin jokaista varmennettavan laitteistokuvauksen rajapintaa kohden. Nämä agentit sisältävät tarvittavat komponentit testistimuluksen vastaanottamiseksi testisekvensseiltä sekä rajapinnan signaalien ajamiseksi ja monitoroimiseksi. Testisekvenssit määrittelevät yksittäisiä testipenkin toimintoja, kuten datatavun kirjoittamisen testattavan moduulin rekisteriin lähettämällä sekvenssipaketin agentin sekvensserin kautta ajurille. Sekvenssipaketti sisältää tiedon siitä, miten ajurin kuuluu ajaa testisyötettä laitteistokuvauksen rajapinnalle. Testit muodostavat kokonaisia testitapauksia aktivoimalla valittuja testisekvenssejä määritellen näin, mitä moduulin toimintoja kyseisessä testissä testataan. Tämän jaon avulla useamman varmennusinsinöörin on helpompi työstää samaa testipenkkiä samanaikaisesti. Abstraktio tarjoaa myös mahdollisuuden uusien testitapauksien luomiseen tuntematta laitteiston rajapinnan matalan tason toimintaa. [2]

Olio-ohjelmoinnin lisäksi UVM tukee komponenttien toiminnan ja kommunikaation erottelemista. Tämä toteutetaan transaction-level modeling (TLM) -porttien avulla

osassa testipenkin komponenttien välisestä kommunikaatiosta. TLM mallintaa laitteistotyyppistä väyläkommunikaatiota korkeammalla abstraktiotasolla. Yksittäisten bittitarkkojen signaalien sijasta TLM-kommunikaatiossa siirretään datan sisältäviä sekvenssipakettiolioita komponentilta toiselle TLM-porttien avulla. Näin mahdollistetaan komponenttien toiminnan ja kommunikaation erottelu, joka lisää komponenttien uudelleenkäytettävyyttä, sillä komponentteja voidaan vaihtaa testipenkin sisällä ilman yhteensopivuusongelmia. [10]

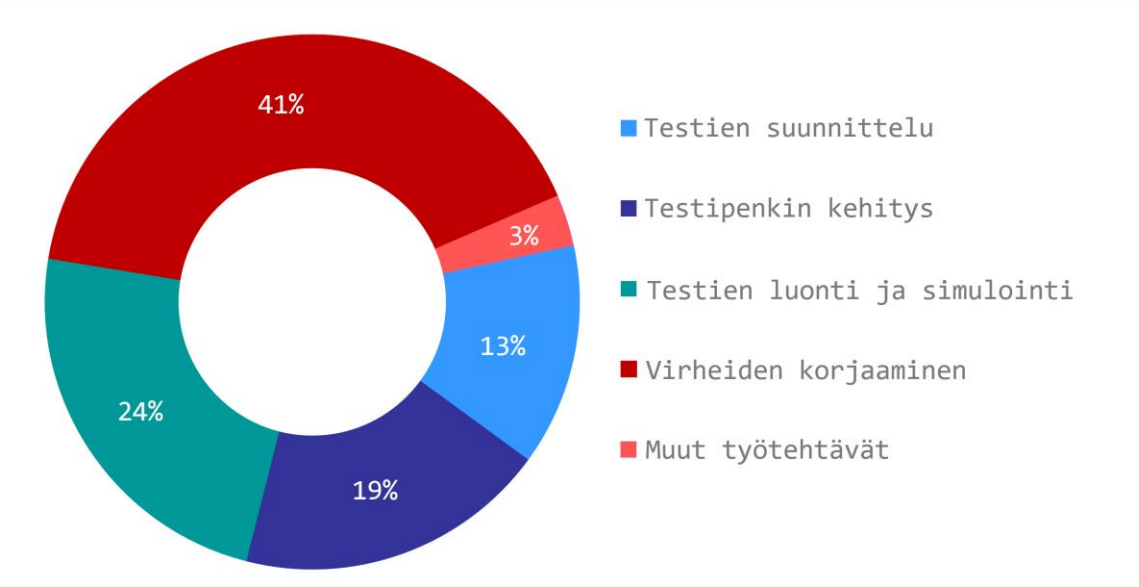
Kuvan 1 mukaisessa UVM-testipenkin hierarkiassa komponenttien väliset nuolet esittävät TLM-kommunikaatiota pois lukien ajurien ja monitorien yhteydet varmennettavaan laitteistokuvaukseen (kuvassa DUV), jotka toteutetaan bittitarkkana väylänä. TLM-yhteydet sekvenssien, sekvenssereiden ja ajurien välillä on sisäänrakennettu UVM-kantaluokkiin, kun taas muiden yhteyksien toteuttaminen on testipenkin kehittäjän vastuulla. [10]

UVM on syrjäyttänyt suurimmaksi osaksi sekä perinteiset laitteistokuvauskielen testipenkit että työkaluvalmistajien omat varmennusmetodit ja se on otettu laajalti käyttöön alan yrityksissä [6]. Koska UVM on erillinen standardi, varmennuksessa ei tarvitse tukeutua yhden työkaluvalmistajan varmennusmetodologiaan, vaan testipenkin kehityksessä voidaan käyttää eri valmistajien työkaluja tarpeen mukaisesti.

UVM-testipenkit voivat sisältää tuhansia lähdekoodirivejä kymmenissä tiedostoissa, joiden kirjoittaminen käsin vaatii aikaa. Vaikka tyypillinen lähtökohta testipenkin kirjoittamiselle onkin käyttää valmista pohjaa tai edellisen projektin testipenkkiä lähtökohtana, lähdekoodia on kirjoitettava paljon käsin. Automatisoimalla testipenkin kehitystä mahdollisimman paljon voitaisiin varmentamisen tehokkuutta parantaa. Tällä tavoin vältyttäisiin suuren lähdekoodimäärän kirjoittamiselta ja työkalulla luodun lähdekoodin virheettömyydestä voitaisiin olla varmoja jo ennen ensimmäistä simulaatiota.

2.3 Menetelmät testipenkkien automaattiseen kehitykseen

Wilson Research Groupin tutkimuksen mukaan laitteistoprojektit käyttivät vuonna 2020 keskimäärin 56 % ajasta varmennukseen. Yksittäiselle laitteistoinsinöörille vastaava luku oli 47 % [6]. Kuvassa 2 esitetään varmennusinsinöörien keskimääräinen suhteellinen ajankäyttö työtehtäviinsä. Kuvasta nähdään, että varmennusinsinöörit käyttävät eniten aikaansa virheiden korjaamiseen, testien luontiin ja simulaatioiden suorittamiseen sekä testipenkin kehittämiseen.



Kuva 2. *Varmennusinsinöörien keskimääräinen suhteellinen ajankäyttö työtehtäviinsä, muokattu lähteestä [6]*

Automatisoimalla testipenkkien kehitystä voitaisiin siihen kuluvaa aikaa lyhentää [11]. Olettamalla työkalun luoman lähdekoodin olevan virheetöntä vähenee myös virheiden korjaamiseen tarvittava aika, sillä virheitä syntyy vähemmän ja virheiden paikantaminen helpottuu. Testien luomiseen, simulaatioiden suorittamiseen ja varmennuksen suunnitteluun vaadittavaan aikaan ei testipenkin kehityksen automatisoinnilla voida olettaa olevan vaikutusta.

Eri laitteistokuvauksia varmentavat UVM-testipenkit sisältävät paljon samankaltaisia rakenteita. Tyypillisesti testipenkkien arkkitehtuuri pysyy verrattain samanlaisena, joten testipenkeissä käytetään samankaltaisia luokkia, joilla on samat tehtävät jokaisessa testipenkissä [2]. Tästä syystä on testipenkkiä luodessa tehokkaampaa lähteä liikkeelle jo olemassa olevasta testipenkistä, vaikka se vaatisikin paljon muokkauksia. Varmennettavalle laitteistokuvaukselle kohdistettua lähdekoodia on kuitenkin aina kirjoitettava testipenkin kehityksen yhteydessä. Tällaisia rakenteita ovat esimerkiksi laitteistokuvauksen rajapinnat, testisekvenssit ja bittitason signaalitoiminta. Vaikka nämä rakenteet eroavat testipenkkien välillä, ovat niiden roolit testipenkin arkkitehtuurissa edelleen samanlaiset.

Testipenkistä toiseen samankaltaisena pysyvä lähdekoodi voidaan luoda työkalun avulla, mutta se voitaisiin yhtä hyvin myös kopioida valmiista mallista tai toisesta testipenkistä. Molemmissa tapauksissa testipenkin ohjelmakoodia on kuitenkin muokattava vähintään nimialueiden törmäysten estämiseksi. Luomalla lähdekoodia työkalun avulla voidaan testipenkkiä muokata joustavasti kirjoittamatta suurta määrää

lähdekoodia uudelleen. Toisaalta tällöinkään ei voida kokonaan välttyä lähdekoodin kirjoittamiselta [12]. Tästä poiketen laitteistokuvauskohtaisten rakenteiden lähdekoodin luominen työkalulla voisi nopeuttaa testipenkin kehitystä. Esimerkiksi laajojen rajapintojen kuvaukset voidaan lukea suoraan laitteistokuvauksesta ja luoda automaattisesti. Lähdekoodin rakenne pysyy samanlaisena, ainoastaan rajapinnan signaalien leveydet ja nimet muuttuvat.

Testipenkin rakenne voidaan kuvata työkalulle yksinkertaisemmalla menetelmällä kuin Systemverilogin kirjoittaminen suoraan. Yhtenä ratkaisuna testipenkki voidaan kuvata yksinkertaisemmassa muodossa merkintäkielellä, jota työkalu käyttää testipenkin lähdekoodin luomiseen [13]. Tämän lähestymistavan haasteena on muodostaa riittävän tarkka kuvaus laitteiston rajapinnan toiminnasta, jotta bittitarkka toiminta voidaan simuloida. Yhtenä mahdollisena ratkaisuna toimintaa ei esitetä bittitarkasti merkintäkielellä, vaan työkalu luo testipenkin arkkitehtuurin ja automatisoitavat rakenteet, ja käyttäjä täydentää testipenkin rakenteisiin osat, joiden luominen työkalulla ei ole kannattavaa. Esimerkiksi testipenkin sekvenssipaketin muuntaminen bittitarkaksi protokollan mukaiseksi liikenteeksi voidaan jättää käyttäjän vastuulle. Standardoitujen rajapintojen varmentamiseen voidaan kuitenkin luoda valmiita varmennuskomponentteja, jolloin vältytään rajapintakohtaisen kuvauksen kirjoittamiselta.

Aiheen aikaisempi tutkimus keskittyy kehittämään testipenkkejä käyttäen perinteisiä laitteistokuvauskieliä [14]. Näiden kielien käyttö varmennustarkoitukseen on vähentynyt huomattavasti alan yrityksissä uudempien varmennusmenetelmien ja -kielien yleistymisen seurauksena [6]. Näiden työkalujen käyttämiä menetelmiä sekä saavuttamia tuloksia on kuitenkin syytä tutkia ennen siirtymistä monimutkaisempiin järjestelmiin.

Vuonna 2016 Shahid Ali Murtza et. al. [11] kehittivät vapaasti saatavilla olevan VertGen-työkalun testipenkkien kehityksen automatisointiin laitteistokuvauksille, jotka käyttävät Verilog-laitteistokuvauskieltä. Työkalu lukee laitteiston portit laitteistokuvauksen tiedostosta, minkä jälkeen käyttäjä syöttää tietoa porttien toiminnasta työkalulle graafisen käyttöliittymän avulla. Työkalu on suunniteltu helppokäyttöiseksi, joten käyttäjältä ei vaadita etukäteistietoa varmennuksesta Verilogia käyttäen. Murtzan et. al. mukaan satunnaistettujen testien kehittäminen on paljon aikaa vaativaa työtä, jota voidaan nopeuttaa hyödyntämällä testipenkin kehityksen automatisointia. VertGen on jälkikäteen integroitu osaksi väitepohjaista varmennusprosessia virheiden tunnistamiseksi laitteistokuvauksesta. [15]

Kai Xian ja Kumar Thulasiraman [16] kehittivät vuonna 2021 avoimen lähdekoodin työkalun monimutkaisuudeltaan keskikokoisten VHDL-laitteistokuvauskielellä kuvattujen laitteistojen testipenkkien kehityksen automatisointiin. Työkalua käytetään samalla tavalla kuin VertGen-työkalua. Työkalu lukee laitteistokuvauksen tiedoston, minkä jälkeen käyttäjä syöttää lisätietoja laitteiston porteista graafisen käyttöliittymän avulla. Kai Xian ja Kumar Thulasiraman mukaan graafinen käyttöliittymä parantaa helppokäyttöisyyden lisäksi myös työkalun joustavuutta, sillä virheiden välttäminen dataa syöttäessä on helpompaa kuin komentoriviä käyttäen. Työkalun luoma testipenkki ei vaadi muokkaamista jälkikäteen, mutta kattavuuslukemat jäävät 60 %:n ja 90 %:n välille eri kattavuusmittareita käyttäen.

Akateeminen tutkimus keskittyy pääosin monimutkaisuudeltaan pienien ja keskikokoisten laitteistokuvauksien varmentamiseen käyttäen perinteisiä laitteistokuvauskieliä. Nämä ratkaisut eivät ole erityisen hyödyllisiä teollisuuden näkökulmasta, sillä modernit digitaaliset järjestelmät ovat usein monimutkaisia ja varmennuksessa käytetään pääasiallisesti Systemverilogiin ja UVM:ään pohjautuvia ratkaisuja. [6]

Kaupallisia ratkaisuja UVM-testipenkkien kehittämisen automatisoimiseksi on kuitenkin olemassa. Suurimmilla työkaluvalmistajilla Siemensilla, Synopyksella ja Cadencella on omat ratkaisunsa. Lisäksi on muiden työkaluvalmistajien ratkaisuja. Näiden työkalujen tavoitteena on nopeuttaa testipenkin kehitystä, madaltaa UVM:n oppimiskynnystä ja helpottaa uudelleenkäyttöä testipenkeissä.

3. VARMENNUS UVM FRAMEWORK -TYÖKALUA KÄYTTÄEN

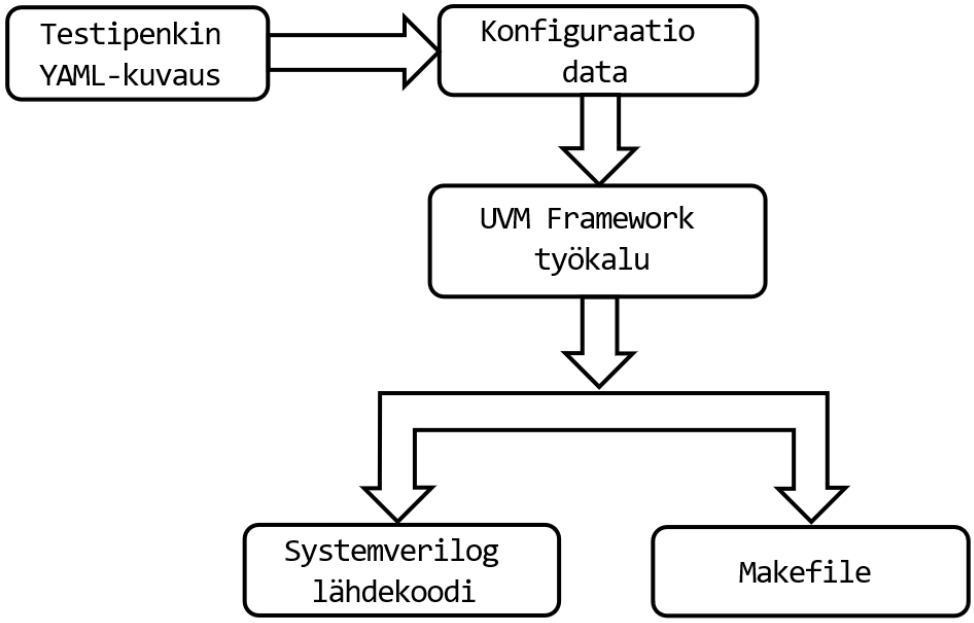
3.1 UVM Framework

Tässä työssä hyödynnettäväksi työkaluksi testipenkin kehityksen automatisointiin valittiin Siemensin UVM Framework ja laitteistosimulaattoriksi Siemensin Modelsim [13]. Siemens on yksi suurimmista alan työkaluvalmistajista, joten työkaluvalinta on perusteltu teollisuudessa tapahtuvan testipenkkien kehityksen esittelyyn. Lisäksi UVM Framework kuuluu simulaattorin toimitukseen, joten erilliselle työkalulisenssille ei ole tarvetta.

Työssä varmennetaan yksinkertainen laitteistokuvaus universaalilla varmennusmenetelmällä työkalua hyödyntäen. Testipenkkiä täydennetään varmennuksen saavuttamiseksi siten, että käsin kirjoitettavan lähdekoodin määrä on mahdollisimman alhainen. Tämän tavoitteena on vähentää varmennukseen vaadittavaa työmäärää mahdollisimman paljon.

Varmennettava laitteistokuvaus on yrityksen omaisuutta, joten siihen liittyvä tieto abstraktoidaan työstä eikä laitteistokuvausta esitellä. Tämä ei vaikuta työn kulkuun, sillä työ keskittyy testipenkin kehityksen ja työkalun esittelyyn. Varmennettava laitteistokuvaus on yksinkertainen, joten simulaatio on sellaisenaan riittävä menetelmä varmennuksen toteuttamiseksi. UVM Framework tukee myös testipenkin emulointia, mutta tämä jätetään pois työstä.

Kuvaus testipenkin rakenteesta luodaan työkalulle käyttäen YAML-merkintäkieltä. YAML on akronyymi merkintäkielen nimestä ”YAML Ain’t Markup Language”. YAML tarjoaa yksinkertaisen tavan kuvata testipenkin rakenne tiivistetysti UVM Framework -työkalulle. Työkalu luo testipenkin tarvittavat komponentit ja tiedostorakenteen kuvauksen perusteella, minkä jälkeen käyttäjä täydentää testipenkkiä käsin.



Kuva 3. *Kaavio UVM Framework -työkalun toiminnasta, muokattu lähteestä [17]*

Testipenkin lähdekooditiedostojen lisäksi UVM Framework luo myös tarvittavat komennot testipenkin ja laitteistokuvauksen kääntämiseen, simulaation suorittamiseen ja aaltomuodon esittämiseen. Nämä komennot työkalu sijoittaa Makefile-tiedostoon komentojen suorittamiseksi Make-työkalulla. Aaltomuodon määrittelevät komennot työkalu sijoittaa simulaattorikohtaiseen tiedostoon, josta simulaattori voi lukea ne simulaatiota aloittaessa. Havainnekuva työkalun toiminnasta esitetään kuvassa 3.

3.2 Testipenkin kuvaaminen

Testipenkin kehittäminen UVM Frameworkin avulla aloitetaan testipenkkiä kuvaavien YAML-tiedostojen kirjoittamisella. Työkalulle kuvataan varmennettavan laitteistokuvauksen rajapinnat, UVM-ympäristö, ympäristön sisäiset komponentit sekä testipenkin ylin taso. Rakenteet voidaan kuvata yhdessä tai useammassa tiedostossa.

Kuvattavan testipenkin komponentti määritellään työkalulle varattujen avainsanojen avulla. Jokainen YAML-tiedosto alkaa avainsanalla `uvmf`, jonka jälkeen muut avainsanat sisennetään kahdella välilyönnillä hierarkian mukaisesti. Avainsanat päätetään kaksoispisteellä. Avainsanojen hierarkian mukainen käyttö on esitelty ohjelmassa 1.

```
uvmf:
  interfaces:
    dut_in:
      clock: clk
      reset: rst_n
      reset_assertion_level: 'False'

      parameters:
      - name: parameter1
        type: int
        value: '1'

      ports:
      - name: port1
        dir: output
        width: '1'
      - name: port2
        dir: input
        width: '1'
      - name: port3
        dir: output
        width: parameter1

      transaction_vars:
      - name: transaction_var1
        type: bit [parameter1-1:0]
        iscompare: 'True'
        isrand: 'True'
```

Ohjelma 1. *YAML-tiedoston aloitus, rajapintojen määrityksen aloitus ja kuvaus rajapinnalle dut_in*

Testipenkin ja laitteistokuvauksen rajapintojen kuvaus aloitetaan avainsanalla `interfaces`. Rajapintojen määrää testipenkissä ei ole rajoitettu. Ensimmäisenä määritellään kello- ja reset-signaalit sekä reset-signaalin polariteetti. Näiden lisäksi määritellään rajapinnan portit sekä sekvenssipaketin sisältö. Lisäksi rajapinnan kuvauksessa voidaan määritellä parametrejä kuvauksen sekä lopullisen testipenkin selkeyttämiseksi.

Työssä käytetyssä laitteistokuvauksessa on kaksi rajapintaa, yksi datan sisääntulolle ja yksi datan ulostulolle. Täten testipenkille päätettiin luoda kaksi rajapintaa, `dut_in` ja `dut_out`. Jokaisesta rajapinnan kuvauksesta muodostuu yksittäinen agentti testipenkin UVM-ympäristöön. Tässä tapauksessa testipenkki kytkeytyy laitteistokuvauksen rajapintaan siis kahden agentin avulla. Yksi datan ajamiseen komponenttiin ja yksi datan lukemiseen komponentista. Rajapinnan `dut_in` kuvaus on esitelty ohjelmassa 1 ja rajapinnan `dut_out` kuvaus ohjelmassa 2. Kuvissa nähdään, että rajapinnoille määritellään parametrit, portit ja sekvenssipakettien sisältö vastaavasti avainsanoilla `parameters`, `ports` ja `transaction_vars`. Parametreille määritellään niiden nimi, tyyppi ja

arvo, porteille niiden nimi, suunta ja leveys ja sekvenssipaketin muuttujille nimi, tyyppi sekä ovatko arvot satunnaistettavissa ja suoraan vertailtavissa UVM:n sisäänrakennetulla funktiolla.

```
dut_out:
  clock: clk
  reset: rst_n
  reset_assertion_level: 'False'

  parameters:
  - name: parameter1
    type: int
    value: '1'
  - name: parameter2
    type: int
    value: '2'

  ports:
  - name: port4
    dir: output
    width: '1'
  - name: port5
    dir: output
    width: parameter2
  - name: port6
    dir: output
    width: '1'
  - name: port7
    dir: input
    width: '1'
  - name: port8
    dir: output
    width: parameter2

  transaction_vars:
  - name: transaction_var1
    type: bit [parameter1-1:0]
    iscompare: 'True'
    israndc: 'True'
  - name: transaction_var2
    type: bit
    iscompare: 'False'
    israndc: 'False'
  - name: transaction_var3
    type: bit [parameter2-1:0]
    iscompare: 'False'
    israndc: 'False'
```

Ohjelma 2. *Kuvaus rajapinnalle dut_out*

Ympäristön sisäisten komponenttien kuvaus aloitetaan avainsanalla `util_components`. Tässä työssä ympäristön sisään luodaan ennustaja, jolla on kaksi TLM-porttia. Yksi `dut_in`-agentille ja toinen `dut_out` -agentille. Ennustajan tehtävänä on muuntaa `dut_in` -agentilta saatavat sisäänmenodataa sisältävät `dut_in_transaction` -sekvenssipaketit

`dut_out_transaction` -sekvenssipaketeiksi, jotta sisäänmenodataa voidaan verrata ulostulevaan dataan pistetaululla käyttäen UVM:n sisäänrakennettua vertailufunktiota. Ennustajakomponentti siis syöttää pistetaululle dataa, jota laitteistokuvauksesta odotetaan luettavan. Tässä työssä luettavan datan oletetaan vastaavan laitteistokuvakselle ajettua dataa, joten ennustajan tehtävänä on ainoastaan datan kopiointi oliosta toiseen. Ennustajakomponentti muodostuu testipenkissä ympäristön sisään. Ympäristön sisäisten komponenttien kuvaus esitetään ohjelmassa 3.

```
util_components:
  dut_predictor:
    analysis_exports:
    - name: dut_in_agent_ae
      type: ’dut_in_transaction #()’
    analysis_ports:
    - name: dut_sb_ap
      type: ’dut_out_transaction #()’
    type: predictor
```

Ohjelma 3. *Ympäristön sisäisten komponenttien kuvaus*

Ympäristön kuvaus aloitetaan avainsanalla `environments`. Ympäristön sisällä määritellään agentit, ympäristön sisäiset komponentit, pistetaulut ja TLM-yhteydet. Testipenkin ympäristön sisään luodaan molempia määriteltyjä rajapintoja vastaavat agentit, ennustajakomponentti sisääntulo- ja ulostulodatan vertailun mahdollistamiseksi, pistetaulu oikean toiminnan tarkastamiseksi sekä TLM-yhteydet agenttien, ennustajan ja pistetaulun välille. Dataa ajavan agentin `dut_in_agent` TLM-portti yhdistetään ensin ennustajaan ja ennustajan TLM-portti pistetauluun. Dataa lukevan agentin `dut_out_agent` TLM-portti yhdistetään suoraan pistetauluun. Näin pistetaulu vastaanottaa kaksi `dut_out_transaction`-sekvenssipakettia, joita voidaan vertailla suoraan oikean toiminnan tarkastamiseksi. Ympäristön kuvaus esitetään ohjelmassa 4.

```
environments:
  dut_env:
    agents:
    - name: dut_in_agent
      type: dut_in
    - name: dut_out_agent
      type: dut_out

    analysis_components:
    - name: dut_pred
      type: dut_predictor

    scoreboards:
    - name: dut_sb
      sb_type: uvmf_in_order_scoreboard
      trans_type: dut_out_transaction

    tlm_connections:
    - driver: dut_in_agent_monitored_ap
      receiver: dut_pred.dut_in_agent_ae
    - driver: dut_pred.dut_sb_ap
      receiver: dut_sb.expected_analysis_export
    - driver: dut_out_agent_monitored_ap
      receiver: dut_sb.actual_analysis_export
```

Ohjelma 4. *Ympäristön kuvaus*

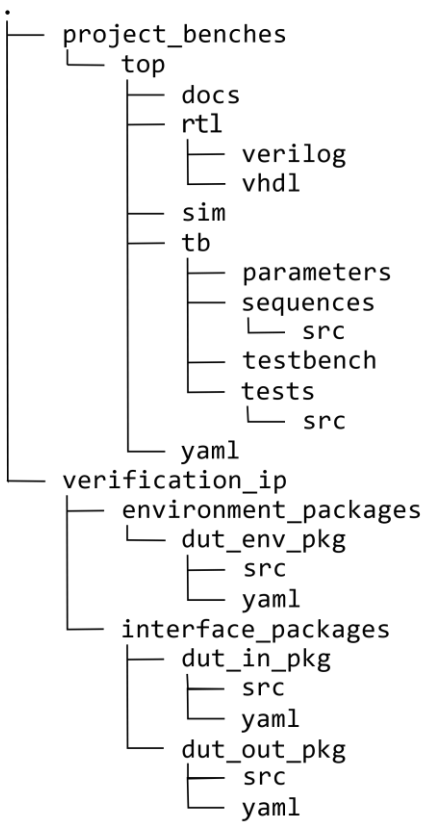
Testipenkin ylimmän tason kuvaus aloitetaan avainsanalla `benches`. Ylimmällä tasolla määritellään agenttien aktiivisuus, kello- ja reset-signaalien ominaisuudet ja korkeimman tason ympäristö. Aktiivinen agentti viestii testattavan laitteistokuvauksen kanssa ajamalla rajapinnan signaaleja. Passiivinen agentti puolestaan ainoastaan monitoroi signaaleja tarkastamista varten ajamatta niitä. Kellosignaalille määritellään puolikas jaksonaika ja vaiheensiirto sekä reset-signaalille aktiivinen taso ja kesto simulaation alussa. Testipenkin molemmat agentit määritellään aktiivisiksi ja kuvattu ympäristö korkeimman tason ympäristöksi. Reset-signaalin aktiivinen taso määritellään loogiseksi nollaksi ja kestoksi valitaan 200 ns. Kellosignaalin jaksonajaksi valitaan 10 ns ja vaiheensiirroksi 9 ns. Kello- ja reset-signaalien kestoilla ei ole vaikutusta simulaatioon, joten arvot valittiin satunnaisesti, kuitenkin niin että asynkroninen reset-signaali ei ole synkroninen kellosignaalin kanssa. Testipenkin ylimmän tason kuvaus esitetään ohjelmassa 5.

```
benches:
  top:
    active_passive:
    - bfm_name: dut_in_agent
      value: ACTIVE
    - bfm_name: dut_out_agent
      value: ACTIVE
    clock_half_period: 5ns
    clock_phase_offset: 9ns

    reset_assertion_level: 'False'
    reset_duration: 200ns

   top_env: dut_env
```

Ohjelma 5. *Testipenkin ylimmän tason kuvaus*

Testipenkin kuvauksen kokonaispituudeksi tuli 118 riviä YAML-lähdekoodia. YAML-merkintäkielen yksinkertaisen ja kompaktin syntaksin ansiosta testipenkin kuvauksen kirjoittaminen on helppoa ja nopeaa. Kuvaus syötetään työkalulle, joka luo testipenkin, kansiorakenteen sekä Makefile-tiedoston testipenkin kääntämiseksi ja simuloimiseksi Make-työkalun avulla.

Kuvassa 4 esitellään työkalun luoma kansiohierarkia. Ainoastaan kansioiden nimet ovat kuvassa näkyvillä. Kuvan kansioissa `docs` on tarkoitettu testipenkin dokumentaatiolle, `rtl` sisältää alikansiot Verilog- tai VHDL-laitteistokuvauskielillä kirjoitetulle varmennettavalle laitteistokuvaukselle. Kansio `sim` sisältää tiedostot Make-työkalun käyttöön kääntämisen ja simuloinnin automatisoimiseksi. Kansiot `yaml` sisältävät YAML-merkintäkielellä kirjoitetut testipenkin kuvauksen osat. Kansiohierarkiassa eri tasoilla sijaitsevat `src` kansiot sisältävät testipenkin lähdekooditiedostot.



Kuva 4. *UVM Framework -työkalun tuottama testipenkin kansiorakenne*

Käytetyn kuvauksen perusteella työkalun luoma testipenkki sisältää 4846 Systemverilog-lähdekoodiriviä. Työkalun luoman testipenkin lähdekoodirivien määrä on noin 39 kertaa suurempi kuin kirjoitetun kuvauksen rivien määrä.

Testipenkin luomisen jälkeen se voidaan kääntää ja simuloida siirtymällä komentorivillä kansioon `sim` kuvan 4 mukaisessa kansiohierarkiassa ja suorittamalla komento `make debug`. Simulaation onnistunut suoritus osoittaa, että työkalun luoma testipenkki toimii oikein. Testipenkki ei kuitenkaan kykene varmentamaan laitteistokuvausta välittömästi testipenkin luomisen jälkeen, vaan sitä on muokattava halutun lopullisen toiminnan saavuttamiseksi.

3.3 Testipenkin täydentäminen

Testipenkin täydentämiseksi muokataan, lisätään tai poistetaan tämän työn tapauksessa 70:tä lähdekoodiriviä 10 tiedostossa ja luodaan 40 uutta lähdekoodiriviä 2:teen uuteen tiedostoon. Lisäksi varmennettava laitteistokuvaus kopioidaan kuvan 4 mukaisessa kansiohierarkiassa kansioon `vhdl`. Laitteistokuvauksen kopiointi testipenkin kansiorakenteeseen ei kuitenkaan ole välttämätöntä, vaan sen polku voi osoittaa minne tahansa tiedostojärjestelmässä.

Varmennettavan laitteistokuvaustiedoston polku määritellään kääntäjälle ja simulaattorille Makefile-tiedostossa, joka sijaitsee kuvan 4 kansiossa `sim`. Polku määritellään muuttujan

```
dut_VHDL_DUT ?= $(UVMF_PROJECT_DIR)/rtl/vhdl/dut.vhd
```

mukaisesti, jossa `dut` on varmennettavan laitteistokuvauksen nimi.

3.3.1 Laitteistokuvauksen instantiointi ja ympäristön konfiguraatio

Testipenkin täydentäminen aloitetaan määrittelemällä laitteistokuvauksen instantiointi. Tämä tapahtuu kansiossa `testbench` tiedostossa `hdl_top.sv`. Instantiointi esitellään ohjelmassa 6. Intanstioitavan laitteistokuvauksen tyyppi on `dut_entity` ja muuttujan nimi on `dut`. Instantioinnissa laitteistokuvaukselle asetetaan mahdolliset parametrit ja yhdistetään portit testipenkin rajapintojen signaaleihin.

```
   dut_entity #( parameter1(1), .parameter2(2) ) dut (
 2     .clk  ( dut_in_agent_bus.clk   ),
       .rst_n( dut_in_agent_bus.rst_n ),
 4     .port1( dut_in_agent_bus.port1 ),
       .port2( dut_in_agent_bus.port2 ),
 6     .port3( dut_in_agent_bus.port3 ),
       .port4( dut_out_agent_bus.port4),
 8     .port5( dut_out_agent_bus.port5),
       .port6( dut_out_agent_bus.port6),
10     .port7( dut_out_agent_bus.port7),
       .port8( dut_out_agent_bus.port8));
```

Ohjelma 6. *Varmennettavan laitteistokuvauksen instantiointi*

Seuraavaksi määritellään ympäristön konfiguraatio. Työn tapauksessa konfiguraatiossa määritellään ainoastaan ympäristön agenttien roolit. Agentin rooli `initiator` tarkoittaa, että agentin ajuri syöttää sekvenssiltä saatavaa testisyötettä varmennettavalle laitteistokuvaukselle. Rooli `responder` tarkoittaa, että agentin ajuri vastaa rajapinnalla näkyvään liikenteeseen. Konfiguraatio määritellään kansion `dut_env_pkg/src` tiedostossa `dut_env_configuration.svh`. Tässä työssä kehitettävässä testipenkissä on kaksi rajapintaa. Rajapinta `dut_in` toimii roolissa `initiator` ja `dut_out` roolissa `responder`. Työkalu määrittelee kaikki agentit automaattisesti rooliin `initiator`, joten ainoastaan `dut_out` rajapinnan rooli täytyy vaihtaa muokkaamalla tiedoston lähdekoodirivi

```
dut_out_agent_config.initiator_responder = INITIATOR;,
```

muotoon

```
dut_out_agent_config.initiator_responder = RESPONDER;.
```

3.3.2 Ennustaja

Ympäristön ennustajakomponentti määritellään tiedostossa `dut_predictor.svh`, joka sijaitsee kansiossa `dut_env_pkg/src`. Komponenttiin täytyy määritellä `dut_in_transaction`-sekvenssipaketin sisältämän datan kopioiminen `dut_out_transaction`-sekvenssipakettiin pistetaulua varten. Työkalu luo ennustajakomponenttiin tulosteet

```
`uvm_info(”UNIMPLEMENTED_PREDICTOR_MODEL”...
`uvm_info(”UNIMPLEMENTED_PREDICTOR_MODEL”...
`uvm_info(”UNIMPLEMENTED_PREDICTOR_MODEL”...
```

ilmaisemaan ei-toteutettua ennustajaa, jotka korvataan lähdekoodirivillä

```
dut_sb_ap_output_transaction.transaction_var1 = t.transaction_var1;,
```

jossa `dut_sb_ap_output_transaction` on `dut_out_transaction` tyyppinen sekvenssipakettiolio ja `t` on `dut_in_transaction` tyyppinen sekvenssipakettiolio. Agentilta `dut_in` saatu data siis kopioidaan `dut_out_transaction`-sekvenssipakettiin, joka siirretään TLM-portin kautta pistetaululle, jotta pistetaulu voi suorittaa vertailun agentilta `dut_out` saadun sekvenssipaketin kanssa. Tässä tapauksessa varmennukseen tarvittava data sijaitsee ainoastaan muuttujassa `transaction_var1`.

3.3.3 Rajapintakomponentit

Työkalun luomassa testipenkissä agenttien sisäiset ajuri- ja monitorikomponentit jakautuvat kahteen osaan: komponentin luokkaan ja rajapintakomponenttiin. Nämä osat sijaitsevat kahdessa tiedostossa. Komponentin luokka kuuluu testipenkin ympäristön luokkahierarkiaan ja rajapintakomponentissa toteutetaan bittitarkan liikenteen käsittely varmennettavan laitteistokuvauksen rajapinnalla. Luokka ja rajapintakomponentti kommunikoivat jaettujen tietorakenteiden avulla. Työkalun luoma luokkakomponentti on sellaisenaan riittävä varmennuksen toteuttamiseksi, joten ainoastaan rajapintakomponenttia tarvitsee muokata.

Agentin `dut_in` ajurin rajapintakomponentti sijaitsee kansion `dut_in_pkg/src` tiedostossa `dut_in_driver_bfm.sv`. Ajuriluokka vastaanottaa sekvenssipaketin sekvensseriltä ja kopioi sen sisällön tietorakenteeseen, josta rajapintakomponentti lukee sisällön. Rajapintakomponentti muuntaa sekvenssipaketin sisällön kuvaaman liikenteen bittitarkaksi väyläprotokollan mukaiseksi liikenteeksi.

Ajurin rajapintakomponentille on määriteltävä rajapinnan signaalien arvot alustustilassa sekä funktio datan muuttamiseksi bittitarkaksi väyläliikenteeksi. Työkalun luoma lähdekoodi rajapinnan signaalien alustukseen esitellään ohjelmassa 7.

```
always @(negedge rst_n_i)
  begin
    // RESPONDER mode output signals
    port2_o <= 'bz;
    // INITIATOR mode output signals
    port1_o <= 'bz;
    port3_o <= 'bz;
  end
```

Ohjelma 7. *Agentin dut_in ajurin rajapintakomponentin signaalien alustus*

Ohjelmassa määritellään ulostulosignaalit rajapintakomponentilta sen molemmissa rooleissa. Koska agentin `dut_in` ajuri on konfiguroitu rooliin `initiator`, asetetaan alustusarvot ainoastaan kyseisen roolin ulostulosignaaleille korvaamalla ohjelman lähdekoodirivit 6 ja 7 lähdekoodiriveillä

```
port1_o <= 'b0;
port3_o <= 'b0;.
```

Ajurin rajapintakomponentti sisältää työkalun luoman funktion `initiate_and_get_response` väyläprotokollan bittitarkkaan toteutukseen. Työkalun funktioon luoma lähdekoodi esitetään ohjelmassa 8. Työkalu luo nämä neljä lähdekoodiriviä kaikkiin neljään testipenkin rajapintakomponentteihin merkitsemään puuttuvaa väyläprotokollan toteutusta.

```
@(posedge clk_i);
@(posedge clk_i);
@(posedge clk_i);
@(posedge clk_i);
```

Ohjelma 8. *Ajurin rajapintakomponentin korvattavat käskyt liikenteen ajamiseksi rajapinnalle*

Työkalun luoma lähdekoodi korvataan väyläprotokollan bittitarkalla toteutuksella, joka esitetään ohjelmassa 9.

```
while (port2_i == 1'b0) @(posedge clk_i);
port1_o <= 1'b1;
port3_o <= dut_in_initiator_struct.port3;
@(posedge clk_i);
port1_o <= 1'b0;
```

Ohjelma 9. *Agentin dut_in ajurin väyläprotokollan bittitarkka toteutus*

Rajapintakomponentti omistaa 2 muuttujaa jokaista rajapinnan signaalia kohden. Muuttujat, joiden nimet loppuvat `_i` ovat sisääntulosignaaleita rajapintakomponentille, joista väylän signaalien arvot luetaan. Muuttujat, joiden nimet loppuvat `_o` ovat ulostulosignaaleita, joiden avulla laitteistokuvauksen portteja ajetaan. Väylälle syötettävä data saadaan ajuriluokalta tietorakenteen `dut_in_initiator_struct` kautta. Funktio odottaa muuttujan `port2_i` saavan arvon 1, minkä jälkeen se syöttää testidataa väylälle rajapinnan protokollan mukaisesti.

Agentin `dut_in` monitorin tehtävä on tarkkailla rajapinnan liikennettä ja luoda `dut_in_transaction`-sekvenssipaketteja, jotka lähetetään ennustajalle TLM-portin avulla. Monitorin rajapintakomponentti `dut_in_monitor_bfm.svh` lukee rajapinnan liikennettä ja välittää luetun datan monitoriluokalle, joka muodostaa datasta sekvenssipaketin. Rajapintakomponentille täytyy määritellä rajapinnan lukeminen funktioon `do_monitor` ohjelman 8 mukaisten lähdekoodirivien tilalle. Korvaava lähdekoodi

```
while (port1_i == 1'b0) @(posedge clk_i);
dut_in_monitor_struct.port3 = port3_i;
```

odottaa muuttujan `port1_i` saavan arvon 1, minkä jälkeen luettu data syötetään monitoriluokalle tietorakenteen `dut_in_monitor_struct` kautta.

Agentin `dut_out` ajurin ja monitorin rajapintakomponentit vaativat samanlaisia muutoksia kuin agentin `dut_in` komponentit, jotta agentti voi suorittaa väyläoperaatioita omalla rajapinnallaan.

Ajurin rajapintakomponentti `dut_out_driver_bfm.sv` sijaitsee kansiossa `dut_out_pkg/src`. Ympäristön konfiguraatiossa agentin `dut_out` ajuriluokka määriteltiin rooliin `responder`, joten se ei reagoi sekvensseriltä saataviin sekvenssipaketteihin vaan rajapinnalla näkyvään liikenteeseen. Rajapinnan signaalien alustusarvot määrittelevä lähdekoodi esitetään ohjelmassa 10.

```
always @(negedge rst_n_i)
  begin
    // RESPONDER mode output signals
    port5_o <= 'bz;
    // INITIATOR mode output signals
    port7_o <= 'bz;
    port8_o <= 'bz;
    port4_o <= 'bz;
    port6_o <= 'bz;
  end
```

Ohjelma 10. *Agentin dut_out ajurin rajapintakomponentin signaalien alustus*

Ohjelman lähdekoodirivi 4 korvataan lähdekoodirivillä

```
port5_o <= ’b0;,
```

joka määrittelee alustusarvot roolin `responder` ulostulosignaaleille. Rajapintakomponentti sisältää työkalun luoman funktion `respond_and_wait_for_next_transfer` väyläliikenteeseen vastaamiselle. Funktion ohjelman 8 mukaiset työkalun luomat lähdekoodirivit korvataan vastauksen määrittelevällä lähdekoodilla

```
port5_o <= 1’b1;
@(posedge clk_i);.
```

Työn tapauksessa rajapintakomponentin ei tarvitse reagoida rajapinnan liikenteeseen vaan signaalin `port5_o` ajaminen loogiseksi ykköseksi koko simulaation ajaksi riittää, sillä monitori suorittaa kaiken liikenteen tulkitsemisen rajapinnalla. Laitteistokuvauksen rajapinnan suorituskykyä ei tarvitse myöskään rajoittaa, sillä monitori kykenee lukemaan kaiken laitteistokuvauksen rajapinnalle ajaman datan.

Agentin `dut_out` monitorin rajapintakomponentin rooli ja toiminta ovat vastaavat agentin `dut_in` monitorin rajapintakomponentin kanssa. Monitorin rajapintakomponentti `dut_out_monitor_bfm.sv` sijaitsee kansiossa `dut_out_pkg/src`. Komponentti sisältää työkalun luoman funktion `do_monitor` rajapinnan tarkkailuun. Funktion ohjelman 8 mukainen työkalun luoma lähdekoodi korvataan rajapinnan liikennettä monitoroivalla lähdekoodilla

```
while(port4_i == 0) @(posedge clk_i);
dut_out_monitor_struct.port6 = port6_i;,
```

joka odottaa signaalin `port4_i` saavan arvon 1, minkä jälkeen se siirtää signaalin `port6_i` arvon tietorakenteeseen `dut_out_monitor_struct`, jonka kautta monitoriluokka saa arvon käyttöönsä.

3.3.4 Sekvenssiluokka

Sekvenssiluokka määrittelee testisekvenssin, jonka perusteella testisyötettä ajetaan laitteistokuvaukselle. Työkalu luo 2 rajapintakohtaista sekvenssiä molemmille rajapintojen agenteille kansioihin `dut_in_pkg/src` ja `dut_out_pkg/src`. Yksi sekvenssi agentin rooliin `initiator`, jolloin sekvenssi ajaa satunnaistettua testisyötettä laitteistokuvaukselle kutsuttaessa sekvenssiä. Toinen sekvenssi on roolille `responder`, jolloin sekvenssi asettaa ajurin rajapintakomponentin odottamaan liikennettä rajapinnalla. Tämän sekvenssin tarkoituksena on vastata rajapinnan liikenteeseen.

Testipenkin ylimmän tason sekvenssit määritellään kansiossa `sequences/src`. Nämä sekvenssit hallitsevat testien suorittamista koko testipenkin tasolla kutsumalla agenttikohtaisia sekvenssejä. Tämän työn tapauksessa luodaan uusi testipenkin sekvenssi `dut_random_sequence` tiedostoon `dut_random_sequence.svh`, joka esitellään ohjelmassa 11. Sekvenssi periytetään työkalun luomasta sekvenssiluokasta `dut_bench_sequence_base`. Koska periytettävä luokka on parametrisoitu, täytyy myös periytyvä luokka määritellä parametrisoiduksi ohjelman rivillä 1. Jättämällä parametrien määrittely kuitenkin tyhjäksi, käyttää luokka periytetyn luokan vakioparametrejä. Sekvenssille luodaan funktio `body`, joka määrittelee sekvenssin toiminnan. Sekvenssi luo ensin käytetyt agenttikohtaiset sekvenssit. Agentille `dut_out` luodaan vastaajasekvenssi ja agentille `dut_in` luodaan satunnaistettu testisekvenssi. Sekvenssi käyttää työkalun agentin `dut_in` konfiguraatioon määrittelemiä metodeja odottaakseen laitteistokuvauksen alustusta sekä lisää 10 kellojakson odotusajan sen jälkeen. Tämän jälkeen sekvenssi aloittaa agentin `dut_out` vastaajasekvenssin, joka jää suorittamaan taustalle, jotta agentin `dut_out` ajuri pysyy päällä koko simulaation ajan. Sekvenssi ajaa satunnaistetun testisekvenssin 25 kertaa ja lisää lopuksi 50 kellojakson mittaisen odotusajan simulaation loppuun varmistaakseen, että kaikki laitteistokuvauksen agentille `dut_out` ajama data on vastaanotettu ja tarkistettu pistetaululla.

```
class dut_random_sequence #() extends dut_bench_sequence_base;

  typedef dut_out_responder_sequence dut_out_responder_sequence_t;
  dut_out_responder_sequence_t dut_out_responder_seq;

  virtual task body();
    dut_out_responder_seq =
      dut_out_responder_sequence_t::type_id::create("dut_out_
      responder_seq");

    dut_in_agent_random_seq =
      dut_in_agent_random_seq_t::type_id::create("dut_in_agent_
      random_seq");

    dut_in_agent_config.wait_for_reset();
    dut_in_agent_config.wait_for_num_clocks(10);

    fork
      dut_out_responder_seq.start(genq001_out_agent_sequencer);
    join_none

    repeat (25) begin
      dut_in_agent_random_seq.randomize();
      dut_in_agent_random_seq.start(dut_in_agent_sequencer);
    end

    dut_in_agent_config.wait_for_num_clocks(50);
  endtask : body
endclass : dut_random_sequence
```

Ohjelma 11. *Testipenkin laajuinen testisekvenssi*

Luotu testisekvenssi on lisättävä sekvenssit sisältävään käännöspakettiin. Käännöspaketti määritellään tiedostossa `dut_sequences_pkg.sv` kansiossa `sequences`. Tiedostoon lisätään lähdekoodirivi

```
`include "src/dut_random_sequence.svh"
```

sekvenssin lisäämiseksi käännöspakettiin.

3.3.5 Testiluokka

Viimeisenä on luotava testiluokka, joka valitsee `dut_random_sequence` sekvenssin käytettäväksi testipenkin testisekvenssiksi. Testiluokka `dut_random_test` esitellään ohjelmassa 12. Testiluokka periytetään työkalun luomasta testiluokasta `test_top`. Metodissa `build_phase` testiluokka ylikirjoittaa kaikki `dut_bench_sequence_base`-luokan instanssit `dut_random_sequence`-luokan instansseilla.

```
class dut_random_test extends test_top;
  `uvm_component_utils(dut_random_test)

  function new(string name = ””, uvm_component parent = null);
    super.new(name, parent);
  endfunction : new

  virtual function void build_phase(uvm_phase phase);
    dut_bench_sequence_base::type_id::set_type_override(
        dut_random_sequence #()::get_type());

    super.build_phase(phase);
  endfunction : build_phase
endclass : dut_random_test
```

Ohjelma 12. *Testipenkin testiluokka*

Luotu testiluokka on lisättävä testiluokat sisältävään käännöspakettiin. Käännöspaketti määritellään tiedostossa `dut_tests_pkg.sv` kansiossa `tests`. Tiedostoon lisätään lähdekoodirivi

```
`include ”src/dut_random_test.svh”
```

Lopullisen testipenkin pituus on 4873 lähdekoodiriviä. Työkalu luo muokkausta vaativiin testipenkin kommentteja ilmaisemaan muokkaamisen tarvetta sekä lähdekoodia, joka on tarkoitus korvata toiminnallisella lähdekoodilla. Koska testipenkkiä muokatessa lähinnä korvataan tai poistetaan työkalun luomaa lähdekoodia, ei testipenkin lähdekoodin määrä juuri muutu muokkauksen aikana.

4. JOHTOPÄÄTÖKSET

Testipenkin kuvausta kirjoitettiin 118 lähdekoodirivin verran ja työkalun luoma ja lopullinen testipenkki eroavat yhteensä 110 lähdekoodirivin kohdalla. Yhteensä lähdekoodia on siis kirjoitettu, poistettu tai muokattu 228 rivin verran. Lopullisen testipenkin pituus lähdekoodiriveinä on 4873, jolloin testipenkin lähdekoodirivejä muokattiin ja lisättiin 2,3 %:a testipenkin lopullisesta pituudesta. Kun YAML-lähdekoodi lasketaan tähän mukaan, on luku 4,7 %.

Koska käytetty laitteistokuvaus on yksinkertainen, pystyttiin varmennus toteuttamaan luomalla suurin osa testipenkin lähdekoodista työkalun avulla. Monimutkaisempien laitteistokuvauksien tapauksessa lähdekoodia pitäisi kirjoittaa enemmän, sillä tarvittaisiin kattavimpia testisekvenssejä ja -tapauksia ja rajapintojen väyläprotokollien bittitarkat toteutukset olisivat monimutkaisemmat.

Tuloksista nähdään, että työkalun avulla kyetään vähentämään lähdekoodin kirjoittamisen tarvetta ja näin ollen varmennukseen vaadittavaa työmäärää ja aikaa. On kuitenkin huomattava, että vaikka kirjoitettavan lähdekoodin määrää voidaan vähentää huomattavasti testipenkin kehityksessä, vaadittava työmäärä ja aika eivät vähene samassa suhteessa. Suurin osa työkalun luomasta lähdekoodista on testipenkistä toiseen toistuvaa lähdekoodia, kun taas ajurit, monitorit ja pistetaulut ovat tyypillisesti vaativimpia komponentteja kehittää vaatien enemmän aikaa ja vaivaa [18]. Myöskin kuten luvussa 2.3 todetaan, voidaan testipenkin kehittämisen automatisoinnin olettaa vaikuttavan ainoastaan työtehtäviin, joihin varmennusinsinöörit käyttävät keskimäärin 60 % ajastaan [6]. Muiden työtehtävien ajankäyttöön automatisoinnin ei voida olettaa vaikuttavan.

Syy testipenkkien kehittämisen automatisoinnin hyödyntämiselle on testipenkkien kehittämiseen vaadittavan työmäärän ja ajan vähentäminen. Tulosten perusteella UVM-testipenkkien automatisoitua kehitystä voidaan pitää mahdollisena menetelmänä varmennuksen työmäärän vähentämiseksi. Varmuuden saavuttamiseksi aihe vaatisi kuitenkin enemmän tutkimusta työkalun hyödyntämisen vaikutuksesta testipenkin kehittämiseen kuluvaan aikaan sekä monimutkaisempien laitteistokuvausten vaikutuksesta tutkimusasetelmaan.

Monimutkaisempien järjestelmien varmennuksessa käytetyt testipenkit voivat olla huomattavasti lähdekoodimäärältään suurempia kuin tässä työssä toteutettu testipenkki. Erityisesti testipenkin ympäristöjen ja agenttien määrän kasvaessa rakenteellisen

lähdekoodin määrä kasvaa huomattavasti. Koska työkalu automatisoi nimenomaan tämän rakenteellisen lähdekoodin kehittämistä, voisi työkalun käyttämisestä hyötyä enemmän laajempien järjestelmien varmennuksessa. Toisaalta testipenkin kehittämiseen kuluvan ajan kasvaessa vähenee työkalun säästämän ajan suhde koko testipenkin kehittämiseen kuluvaan aikaan. Tällöin on arvioitava, onko työkalun tarjoama hyöty suurempi kuin sen mukanaan tuomat heikkoudet.

Työssä käytetyllä työkalulla on myös heikkouksia käsin kirjoitettuun testipenkkiin verrattuna. UVM Framework on tarkoitettu käytettäväksi sekä yksittäisten laitteistokuvausten että laajempien järjestelmien testipenkkien kehitykseen. Koska työssä varmennettava laitteistokuvaus on yksinkertainen, sisältää työkalun luoma testipenkki ylimääräisen monimutkaisia rakenteita. Esimerkiksi ajurien ja monitoreiden jakaminen luokka- ja rajapintakomponentteihin emulaation tukemiseksi ei ole tarpeellista, kun varmennus toteutetaan käyttäen pelkästään simulaatiota. UVM Frameworkin luoma testipenkki lisäksi periyttää kaikki testipenkin luokat työkalukohtaisista kantaluokista UVM-kantaluokkien sijaan. Tämä abstraktoi testipenkin toimintaa pois käyttäjän näkyvistä. Tämä vähentää tarvittavaa kokemusta UVM-testipenkkien kehityksestä, mutta vähentää testipenkin joustavuutta sekä vaatii, että UVM Framework on asennettuna, jotta testipenkkiä voidaan kehittää ja käyttää. Lisäksi työkalun luoma lähdekoodi kärsii käsin kirjoitettua lähdekoodia huonommasta luettavuudesta ja ymmärrettävyydestä.

5. YHTEENVETO

Tämän työn tarkoituksena on selvittää, miten laitteistokuvausten varmentamisessa voidaan hyötyä UVM-testipenkin automatisoidusta kehityksestä. Tavoitteena automatisoinnissa on vähentää testipenkin kehitykseen vaadittavaa työmäärää sekä nopeuttaa testipenkkien kehitystä.

Testipenkkien kehityksen automatisointiin on olemassa useita sekä vapaasti saatavia että kaupallisia työkaluja, mutta suurin osa tieteellisestä tutkimuksesta ja vapaasti saatavilla olevista työkaluista keskittyy perinteisillä laitteistokuvauskielillä tehtyjen testipenkkien kehittämiseen. Universaalia varmennusmenetelmää käyttävät työkalut taaskin ovat lähes yksinomaan kaupallisia.

Tässä työssä demonstroidaan Siemensin UVM Framework -työkalua testipenkin kehityksessä yksinkertaiselle laitteistokuvaukselle. Hyödyntämällä työkalua voidaan kirjoitettavan lähdekoodin määrää vähentää huomattavasti, mutta lähdekoodin laatu kärsii käsin kirjoitettua testipenkkiä huonommasta luettavuudesta sekä ylimääräisestä monimutkaisuudesta ja abstraktiosta.

Vaikka työssä onnistuttiin vähentämään käsin kirjoitettavan lähdekoodin määrä, tulosten perusteella ei kuitenkaan voida sanoa onko työkalun käytöllä merkittävää vaikutusta testipenkin kehittämiseen kuluvaan aikaan. Tulokset voivat myös vaihdella käytettäessä monimutkaisempaa varmennettavaa laitteistokuvausta sekä testipenkin koon kasvaessa käytettäessä useampia UVM-agentteja ja -ympäristöjä.

Jatkotutkimuksena työn pohjalta voisi tutkia, miten testipenkin kehitystä automatisoivan työkalun käyttö vaikuttaa testipenkkiin monimutkaisemman laitteistokuvauksen tapauksessa. Korkeamman laitteistokuvaushierarkian kohdalla testipenkin koko voisi kasvaa huomattavasti, mikäli testipenkki sisältäisi useampia UVM-ympäristöjä laitteistokuvauksen sisäisten uudelleenkäytettyjen moduulien tarkkailuun. Tällöin työkalun luoman lähdekoodin määrä kasvaisi, mikä vähentäisi käsin kirjoitetun lähdekoodin suhdetta testipenkin kokoon. Samoin työkalun vaikutusta testipenkin kehittämiseen kuluvaan aikaan voisi tutkia paremman kuvan saavuttamiseksi automatisoidun kehityksen hyödyistä. Tällöin testipenkin kehitykseen kuluvaa aikaa voisi verrata esimerkiksi ajankäyttöön aloitettaessa testipenkin kehitys olemassa olevan testipenkin pohjalta.

LÄHTEET

[1] Foster, Harry. Why the design Productivity Gap Never Happened. Proceedings of the International Conference on Computer-Aided Design. IEEE Press, Nov 2013, pp. 581-584.

[2] IEEE Standard for Universal Verification Methodology Language Reference Manual. IEEE, Std 1800.2-2017, Sep 2020.

[3] Henkel, Jörg. Closing the SoC design gap. Computer Volume 36 Issue 9. Sep 2003, p119–121.

[4] Wilson, Ron. Help may be en route for SoC verification, power. Electronic Engineering Times, Issue 1240. Oct 2002, p64-66.

[5] Ashar, Pranav & Viswanath, Vinod. Closing the Verification Gap with Static Sign-off. 20th International Symposium on Quality Electronic Design (ISQED), Mar 2019, p343–347.

[6] Foster, Harry. The 2020 Wilson Research Group Functional Verification Study. Siemens corporation, Feb 2021, saatavissa (viitattu 10.3.2022): https://blogs.sw.siemens.com/verificationhorizons/2021/02/10/conclusion-the-2020-wilson-research-group-functional-verification-study/

[7] Bergeron, Janick et al. Verification Methodology Manual for SystemVerilog. 1st ed. Springer US, New York, NY, 2006.

[8] Fiergolski, A. Simulation environment based on the Universal Verification Methodology. Journal of instrumentation 12, Jan 2017.

[9] Vineeth B, Tripura Sundari BB. UVM Based Testbench Architecture for Coverage Driven Functional Verification of SPI Protocol. In 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2018. p. 307–310.

[10] Universal Verification Methodology 1.2 User's Guide. Accellera, Oct 2015, Saatavissa (viitattu 14.05.2022): https://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf

[11] Murtza, Shahid Ali et al. VerTGen: An automatic verilog testbench generator for generic circuits. 2016 International Conference on Emerging Technologies (ICET). IEEE, Oct 2016.

[12] Edelman, Rich & Bhutada, Shashi. UVM Sans UVM: An approach to automating UVM testbench writing. Siemens Verification Academy, 2015, saatavissa (viitattu 12.05.2022): https://s3.amazonaws.com/verificationacademy-news/DVCon2015/Papers/dvcon-2015_UVM-Sans-UVM-Automating-UVM-Testbench-Writing-Paper.pdf

[13] UVM Framework, Siemens Verification Academy, saatavissa (viitattu 12.05.2022): https://verificationacademy.com/topics/verification-methodology/uvm-framework

[14] Vivekananda, Ashish Alape & Enoiu, Eduard. Automated Test Case Generation for Digital System Designs: A Mapping Study on Vhdl, Verilog and Systemverilog Description Languages. Designs 4.3. Sep 2020, pp 1-19.

[15] Murtza, Shahid Ali et al. AAG: An automatic assertion generation framework for RTL designs, 2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET). IEEE, Mar 2018.

[16] Kai Xian, Kenneth Tan & Kumar Thulasiraman, Nandha. An Automatic VHDL Testbench Generator for Medium Complexity Design. In 2021 IEEE 19th Student Conference on Research and Development (SCOReD). IEEE, Nov 2021, pp 113-118.

[17] Sühnel Christoph. Making it Easy to Deploy the UVM. Verification Horizons Volume 9, Issue 2. Siemens Verification Academy, Jun 2013, saatavissa (viitattu 11.05.2022): https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-9_issue-2/articles/stream/making-it-easy-to-deploy-the-uvm_vh-v9-i2.pdf

[18] Horn, Michael et al. UVM Cookbook. Siemens Verification Academy, 2018, saatavissa (viitattu 13.05.2022): https://verificationacademy.com/cookbook/uvm