TIETO- JA SÄHKÖTEKNIIKAN TIEDEKUNTA

ELEKTRONIIKAN JA TIETOLIIKENNETEKNIIKAN TUTKINTO-OHJELMA

KANDIDAATINTYÖ

SystemVerilog-assertioiden formaali tarkastus

Tekijä Joonatan Helminen

Ohjaaja Jukka Lahti

Huhtikuu 2023

Helminen J. (2023) SystemVerilog -assertioiden formaali tarkastus. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 25 s.

TIIVISTELMÄ

Tässä kandidaatin työssä perehdytään, miten SystemVerilog-laitteistokuvauskielessä väitteiden formaali tarkastus toimii käyttämällä automaattista formaali tarkastustyökalu OneSpin 360 Design Verification (DV)-ohjelmaa. Työn sisällössä perehdytään aluksi teoriapuolella SystemVerilog -laitteistokuvauskieleen ja siinä esiintyviin väitteisiin, sekä formaaliseen tarkastukseen ja sen käyttämiseen väitteiden tarkastuksessa. Käytännön puolessa käyttämällä OneSpin 360 DV-ohjelmaa apuna tutkitaan, miten väitteiden formaali tarkastus toimii. Lopuksi käsitellään aihetta sekä pohditaan ohjelmasta saatuja tuloksia.

Avainsanat: SystemVerilog-väite, Formaali tarkastus.

Helminen J. (2023) Formal verification of SystemVerilog assertions. University of Oulu, Degree Programme in Electronics and Communications Engineering, Bachelor Thesis, 25 p.

ABSTRACT

In this bachelor's thesis, the formal verification of assertions in the SystemVerilog Hardware Description Language is explained and verified using the automatic formal verification tool OneSpin 360 Design Verification (DV) program. At first, the theory about SystemVerilog Hardware Description Language, assertions in it, formal verification, and its use for verification of assertions are explained. Then in the practical side, using the OneSpin 360 DV program as an aid, formal verification of assertions is verified. In the end, the topic is covered, and the results obtained from the program are discussed.

Key words: SystemVerilog Assertion, Formal verification.

SISÄLLYSLUETTELO

LYHENTEIDEN JA MERKKIEN SELITYKSET

SV	SystemVerilog
HDL	Hardware Description Language, Laitteistokuvauskieli
HVL	Hardware Verification Language, Laitteistoverifiointikieli
HDVL	Hardware Description and Verification Language, Laitteisto kuvaus- ja verifiointikieli
DUT	Design Under Test, Suunnittelu testattavana
OOP	Object Oriented programming, Olio-ohjelmointi
SVA	SystemVerilog Assertion Language, SystemVerilog-väitekieli
API	Application Programming Interface, Ohjelmointirajapinta
FV	Formal Verification, Formaali tarkastus
IC	Integrated Circuit, Integroitu piiri
RTL	Register Transfer level, Rekisterisiirtotaso
GLS	Gate Level Simulation, Porttitasosimulaatio
LEC	Logical Equivalence Checking, Loogisen vastaavuuden tarkistus
SEC	Sequential Equivalence Checking, Peräkkäisen vastaavuuden tarkistus
APB	Advanced Peripheral Bus, Edistyksellinen oheisväylä
SOC	System-On-Chip, Järjestelmä sirulla
M	System Model, Järjestelmämalli
ϕ	Property, Ominaisuus

1 JOHDANTO

SystemVerilog laitteistokuvaus- ja laitteistoverifiointikieli on laajalla alueella käytössä puolijohde- ja elektroniikkasuunnitteluteollisuudessa sen laitteiston kuvaus ja verifiointi ominaisuuksien takia. SystemVerilog-kielessä esiintyvät väitteet sekä formaalia tarkastusta käyttävät työkaluohjelmat ovat osa laitteistojen suunnittelun tarkastusta. Tässä kandidaatintyössä on tavoitteena oppia, perehtyä ja toteuttaa SystemVerilog-kielessä esiintyvien väitteiden formaali tarkastus.

Tämä kandidaatintyö koostuu teoria- ja toteutusosioista. Teoriaosuudessa perehdytään SystemVerilog-kielen, kielessä esiintyviin väitteiden, formaaliin tarkastuksen, sekä väitteiden formaalin tarkastuksen merkitykseen laitteiden suunnittelussa ja tarkastuksessa. Teorian pohjalta siirrytään toteutusosioon, jossa käytännönpuolen työnä tehdään Digitaalitekniikka 3-kurssin X2Go-etätyöpöytäympäristössä Linux-palvelimella väitteiden formaali tarkastus käyttäen apuna automaattista formaali tarkastustyökalu OneSpin 360 DV-ohjelmaa. Sen jälkeen pohditaan ohjelmasta saatujen tulosten ja teorian pohjalta, väitteiden ja formaalin tarkastuksen käyttämistä suunnitelman oikeellisuuden varmentamiseen.

2 AIHEEN KÄSITTELY

2.1 SystemVerilog

SystemVerilog (SV) on Verilog laitteistokuvauskielen (Engl. Hardware Description Language) laajennus monilla tarkastusominaisuuksilla, jotka mahdollistavat suunnittelun tarkistamisen käyttämällä monimutkaisia testipenkkirakenteita ja satunnaisia ärsykkeitä simulaatiossa [1][2][3]. Laitteistokuvauskieltä (HDL), kuten Verilog, käytetään kuvaamaan laitteiston käyttäytymistä, jotta se voidaan muuntaa digitaalisiksi lohkoiksi, jotka koostuvat kombinaatiologiikkaporteista ja peräkkäisistä sekvenssilogiikkakomponenteista [3]. SV-kieli sisältää yhdistämällä rinnakkain laitteistokuvauskielen sekä laitteistoverifiointikielen (Engl. Hardware Description and Verification Language, HDVL) siten, että ne on yhdessä suunniteltu toimimaan ja parantamaan toistensa puuttuvia ominaisuuksia [1]. HDL pitää tarkistaa oikeaksi, joten tarvitaan enemmän tarkastusominaisuuksia sisältävä laitteistoverifiointikieli (HVL), joka tukee monimutkaisia testausprosesseja. SV-kieli on mahdollistanut HDVL käytön, jota pystytään käyttämään abstraktisten ja yksikohtaisten laitteiden suunnitteluun ja mallintamiseen, sekä kattavuuden, testipenkkitarkastuksien ja simulointien seuraamiseen manuaalisia tai automaattisia menetelmiä käyttäen [1].

Verilog-kieli voi tarkistaa vain ensisijaisesti pienten, ei kovin monimutkaisten ja vähemmän ominaisuuksia sisältävien laitteiden mallien toimivuuden ja testauksen rekisterisiirtotasolla (Engl. Register Transfer Level, RTL) [2]. Kun suunnittelun monimutkaisuus ja ominaisuudet lisääntyvät tarvitaan parempia työkaluja suunnitteluun ja tarkistamiseen, jonka SV-kieli suorittaa käyttämällä rajoitettuja satunnaisia ärsykkeitä, OOP ominaisuuksia testipenkin rakentamisessa, toiminnallisuuden kattamisessa, sekä väitteitä oikeellisuuden varmistamisessa [1][3]. SV-kieli tukee näillä ominaisuuksilla portti-, rekisteri- ja käyttäytymistasoja, joten voidaan ajatella, että sitä käyttämällä pystytään kokonaan suunnittelemaan, mallintamaan, simuloimaan, testaamaan ja toteuttamaan elektronisia järjestelmiä.

2.2 SystemVerilog-kielen väitteet

Väitteet (Engl. Assertions) ovat osa SV-kieltä, joista on muodostunut SVA-kieli (Engl. SystemVerilog Assertion Language), jossa ne ovat formaaleja lausekkeita, joita pystytään käyttämään tarkastamaan, että vaadittu tai oletettu toiminta toteutuu digitaalisessa piirissä, kun piirille syötetty tieto sekä kriteerit ovat piirin toiminnan mukaisia [1][4][5]. Väite on positiivinen toteamus suunnittelumallin ominaisuudesta (Engl. Property), koska sen on tarkoitus olla tosi aina. Ominaisuus ilmaisee vain käyttäytymisen, joten sitä käytetään usein varmistamaan, että käyttäytymisen suunnittelutoteutus vastaa väitettä, eli jos väite todetaan vääräksi, se osoittaa virheen suunnittelussa. SV-kieli mallintaa käyttäytymistä eksplisiittisten aika-alueiden ominaisuuksilla, jotka ovat synkronisia kellojen kanssa sekä asynkronisia tietyillä aikaviiveillä. SVA-kielessä väitteitä voidaan käyttää suunnitteluvirheiden, kuten virheellisten logiikka-arvojen, ajoitusrikkomusten tai virheellisen tiedonkulun, havaitsemiseen ja raportoimiseen. Niitä voidaan myös käyttää protokollarikkomusten tai muiden odottamattomien olosuhteiden tarkistamiseen. Väitetekniikka kehitettiin, jotta suunnitteluspesifikaatioiden kirjoittaminen on formaalisesti ratkaisevassa roolissa suunnitteluvirheiden havaitsemisessa, jolloin tarkastustyökalut voivat automaattisesti havaita virheet toteutetun suunnittelun ja sen spesifikaatioiden perusteella [3]. Väitteitä pystytään tarkistamaan joko simulaattorin avulla, jossa simuloinnin yhteydessä seurataan, toteutuuko

väitteet sen aikana, tai formaali verifiointiohjelmalla tarkastaa piirin mallia tutkimalla staattisesti toteutuvatko väitteet vai ovatko ne vääriä [5].

SV-kielessä väitteiden tarkistamiseen käytetään erilaisia tarkistusmenetelmiä. Väitteillä tarkistetaan ominaisuuslausetta. Alla on seuraavat tarkistusmenetelmät, jossa ominaisuus on suunnitelman porttien tai sisäisten muuttujien arvoista laskettu looginen lauseke, tai loogiset lausekkeet useita kellojaksoja kestävää sekvenssiä [5]:

- Assert-väite tarkistaa ominaisuuslausetta, jossa ominaisuuden tulee olla aina totta, eli todetaan, että tietyn ominaisuuden toteutuminen on pakollinen ehto, jotta piirin oikea toiminta toteutuu.
- Assume-väite tarkistaa ominaisuuslausetta, jossa ominaisuuden oletetaan olevan aina totta, jonka validiteettisimulaattori tarkistaa, joka kertoo, että tietty ominaisuus on oletus piirin toimintaympäristöstä, jota formaaliset varmennusohjelmat käyttävät oletusarvoisesti todistaessaan väitteitä oikeiksi tai vääriksi.
- Cover-väite tarkistaa ominaisuuslausetta, jossa ominaisuus voi olla joskus totta, ja jolla seurataan, kuinka usein tietty ominaisuus toteutui tarkistuksessa tai simulaatiossa, testausärsykkeiden kattavuuden arvioimiseksi.
- Restrict-väite tarkistaa ominaisuuslausetta, jota voidaan käyttää formaalisessa tarkastuksessa määrittämään jokin ominaisuus rajoitukseksi.

SVA-kielessä väitteitä on kahta tyyppiä, joko välittömiä väitteitä (Engl. Immeadiate Assertion) tai konkurrenttisia väitteitä (Engl. Concurrent Assertion). Välittömät väitteet ovat loogisia ja kellosta riippumattomia, jotka sijoitetaan proseduraalisen koodin sisälle, jossa ne suoritetaan normaaliin tapaan aina, kun kyseinen lause aktivoituu simuloinnin aikana samalla tavalla kuin if-lauseet proseduraalisessa koodissa. Välittömät väitteet on tarkoitettu käytettäväksi simulaatiossa, eivätkä ne sovellu formaaliseen tarkastukseen, koska välitön väite onnistuu, jos lauseke pitää paikkansa käskyn suoritushetkellä, ja epäonnistuu, jos lausekkeen arvo on epätosi, kuten X, Z tai 0. Konkurrenttiset väitteet tarkkailevat koko ajan kellon mukaan varmennettavan mallin suunnitelmaa sen rinnalla toteutumalla erillisinä prosesseina jatkuvasti, jossa assert-, assume-, cover tai restrict-väite sijoitetaan moduulissa proseduraalisen osan ulkopuolelle, kuten toiseen moduuliin, jolloin syntyy konkurrentti väite. Konkurrenttiset väitteet tarkastavat suunnitelman jonkin ominaisuuden toiminnan ja voimassaolon, jotka kuvataan ominaisuuslauseella, jolloin niiden käyttö sopii formaaliseen tarkastukseen [5][6][7].

Seuraavassa taulukossa 1. esitetään yleisimpiä lausekkeita (Engl. Expressions) SVA-kielestä, joita käytetään yhdistelmä Boolen kaavojen (Engl. Boolean Formulas) rakentamiseen käyttämällä tavallisia Verilog-kielen Boolen konnektiiveja, kuten AND, OR ja NOT, kun taas operaattorit (|->, |=>) ottavat Boolen lausekkeet ja muuntavat ne ajallisiksi ominaisuuksiksi, jotka voivat ilmaista ajasta riippuvaa käyttäytymistä.

Taulukko 1. Yleisimmät Lausekkeet ja Operaattorit SVA-kielessä [5]

SVA lausekkeet	Selitys	SVA Operaattorit	Selitys
A && B	A:n ja B:n looginen ”ja”	A |-> B	Kun A pitää, B:n pitää olla voimassa samalla kellojaksolla
A || B	A:n ja B:n looginen ”tai”	A |=> B	Kun A pitää, B:n pitää olla voimassa seuraavalla kellojaksolla
A == B	A ja B loogisesti yhtä suuret	##N	Lauseke pätee N kellojakson jälkeen, esimerkiksi: ##1 (Lausekkeella yhden kellojakson viive) ##2 (Lausekkeella kahden kellojakson viive)
A != B	A ja B loogisesti erisuuret	$rose(A)	Palauttaa arvon tosi, jos lausekkeen A arvon vähiten merkitsevä bitti muuttui 1:ksi edelliseltä kellojaksolta nykyiselle siirryttäessä. Muulloin palauttaa arvon epätosi.
!A	A:n looginen ”epätosi”	$fall(A)	Palauttaa arvon tosi, jos lausekkeen A arvon vähiten merkitsevä bitti muuttui 0:ksi edelliseltä kellojaksolta nykyiselle siirryttäessä. Muulloin palauttaa arvon epätosi.

Kuvassa 1. on kaksi yksinkertaista esimerkkiä assert-väitteestä SVA-kielessä. Niiden toiminnallisuus on toteutettu always_ff -proseduurin sisällä riveillä 7–18. Väitteet tarkistavat jokaisella kellon nousevalla reunalla, että ulostuloon (C_OUT) sijoitetaan oikea arvo riippuen sisääntulosignaaleista. Jos väite ei pidä paikkaansa, annetaan virheilmoitus, että ei sijoitettu oikeaa arvoa ulostuloon. Nämä väitteet voidaan tarkistaa simulaatiolla tai formaalisella varmennustyökaluilla.

```
1 module assertion_test
2   (input logic clk,
3    input logic rst_n,
4    input logic a_in,
5    input logic b_in,
6    output logic c_out);
7   always_ff @(posedge clk or negedge rst_n)
8        begin
9         if (!rst_n)
10         c_out <= '0;
11        else
12         begin
13          if (a_in)
14           c_out <= a_in;
15          else if (b_in)
16           c_out <= b_in);
17         end
18       end
19  //assertions
20  property r_follow_a_in;
21        @(posedge clk) disable if (!rst_n)
22          a_in |=> (c_out == a_in);
23  endproperty;
24  property r_follow_a_in;
25        @(posedge clk) disable if (!rst_n)
26          b_in |=> (c_out == b_in);
27  endproperty;
28  ar_follow_a_in: assert property (r_follow_a_in)
29        else $error("c_out != a_in.")
30  ar_follow_b_in: assert property (r_follow_b_in)
31        else $error("c_out != b_in.")
32 endmodule
```

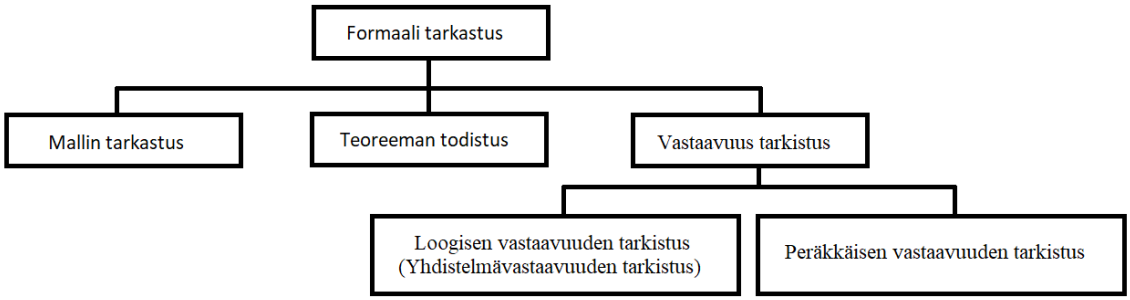
Kuva 1. Väitteet SVA-kielessä

2.3 Formaali tarkastus

Formaali tarkastus (Engl. Formal Verification, FV) on prosessi, jossa sitä käyttävät työkalut analysoivat matemaattisesti erilaisia algoritmeja käyttämällä SystemVerilog-kielellä kirjoitetun suunnitelman mahdollisten käyttäytymismallien tilat. FV on automaattinen tarkistusmenetelmä, jota käytetään osoittamaan, että suunnittelu toimii odotetulla tavalla, sekä täyttää sen vaatimukset ja paljastaa suunnittelun epäselvyydet ja suunnitteluvirheet. FV tarkastaa kokonaan mahdolliset testitapaukset eikä vain yksittäisiä tiettyjä arvoja, mutta se ei suorita kaikkia mahdollisia testitapauksia, vaan käyttää älykkäitä matemaattisia tekniikoita arvioidakseen kaikki tarvittavat mahdolliset käyttäytymismallit. FV-työkalut eivät vaadi ärsykkeitä eikä testipenkkiä, joten tarkastus tehdään IC-suunnittelusyklin varhaisessa vaiheessa heti kun RTL-koodi on saatavilla. Kun tarkastuksen aikana havaitaan vika, se voidaan helposti korjata. Formaali tarkastus nousi suosioon, kun Intel-prosessoreissa löydettiin vuonna 1994 Pentium FDIV-laitteistovirhe. Virheen takia Intelin mikroprosessorit palautti vääriä binääriliukulukutuloksia jakaessaan tiettyjä erittäin tarkkoja lukupareja. Monet laitteistosovellukset ovat välttämättömiä, joissa mikä tahansa vika voi aiheuttaa merkittäviä taloudellisia tai fyysisiä vahinkoja, joten FV nähdään välttämättömänä työkaluna virheiden tunnistamisessa [8].

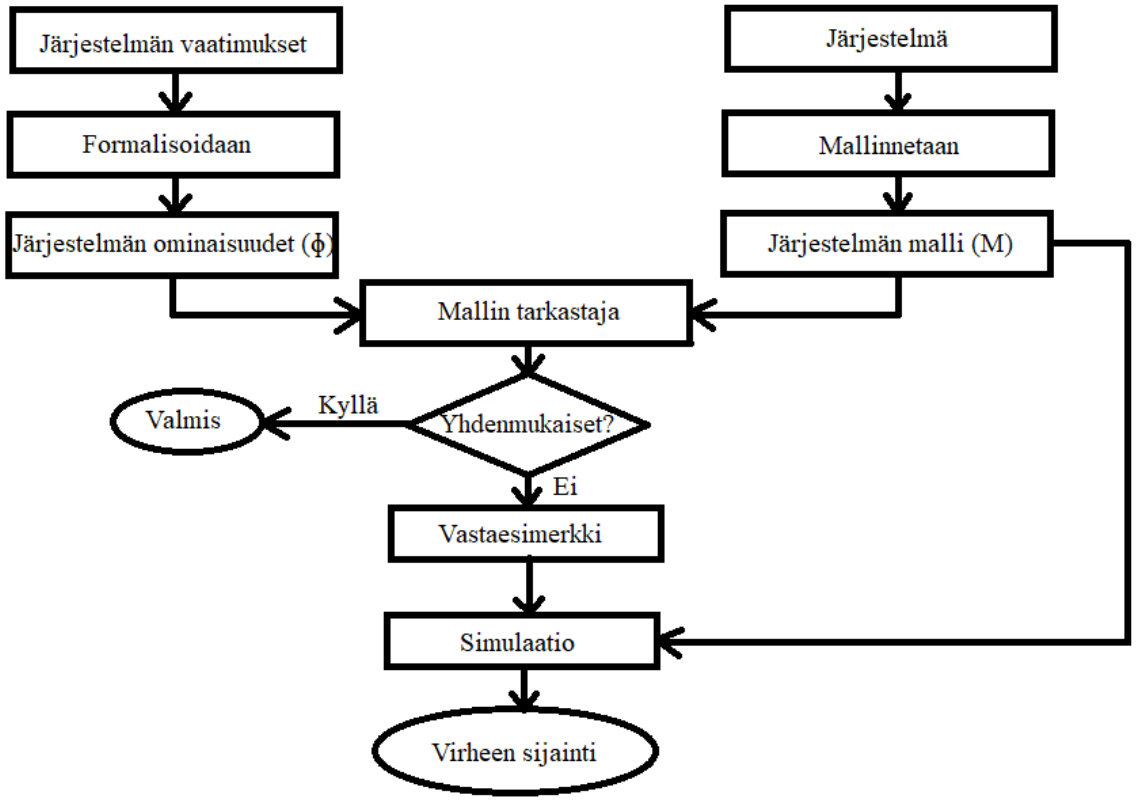
Formaaliset tarkistustekniikat jäljittävät vikoja, kuten "Corner case"-tapauksia, joita ei havaita tavallisilla tarkastustekniikoilla, kuten simulaatiolla. FV tunnistaa viat huomattavasti nopeammin kuin ne voidaan havaita käyttämällä standarditekniikoita. Ennen kuin mallin toiminta varmistetaan simuloinnilla ja emuloinnilla, se varmistetaan formaalisesti. Jotkut formaalisen tarkastuksen edut ovat, että se havaitsee virheet suunnittelusyklin varhaisessa vaiheessa ennen kuin testitiedosto on kehitetty, sitä voidaan käyttää suunnittelun korkean tason, RTL- tai GLS-esityksissä, sen käyttäminen on vähemmän aikaa vievää virheiden löytämiseen, se on luotettava ja nopeampi kuin tavalliset tarkastustekniikat, sekä se on tyhjentävä testaustapa eli se käyttää testaukseen kaikkia mahdollisia datayhdistelmiä [9].

Formaalisessa tarkastuksessa on eri tekniikoita suunnittelun tarkastusta varten, jotka ovat esitetty kuvassa 2. Tekniikat ovat mallin tarkistus (Engl. Model Checking), teoreeman todistus (Engl. Theorem Proving), sekä vastaavuus tarkistus (Engl. Equivalence Checking). Mallin tarkistus on tilaperusteinen lähestymistapa formaaliseen tarkatukseen, joka tunnetaan myös nimellä ominaisuus tarkistus (Engl. Property Checking), jossa järjestelmä mallinnetaan joukkona tiloja. Teoreeman todistaminen on prosessi, jolla varmistetaan, että toteutettu järjestelmä tai suunnittelu täyttää suunnitteluvaatimukset, käyttämällä matemaattista päättelyä. Teoreeman todistaminen on todisteisiin perustuva (Engl. Evidence-Based) lähestymistapa formaaliseen tarkastukseen. Vastaavuus tarkistus on prosessi, jolla varmistetaan, että kaksi mallia ovat toiminnallisesti samanlaisia. Vastaavuuden tarkistustekniikoita on kahden tyyppistä, joista ensimmäinen on loogisen vastaavuuden tarkistus (Engl. Logical Equivalence Checking), joka tunnetaan myös nimellä yhdistelmävastaavuuden tarkistus (Engl. Combinational Equivalence Checking), sekä toinen on peräkkäisen vastaavuuden tarkistus (Engl. Sequential Equivalence Checking) [9].



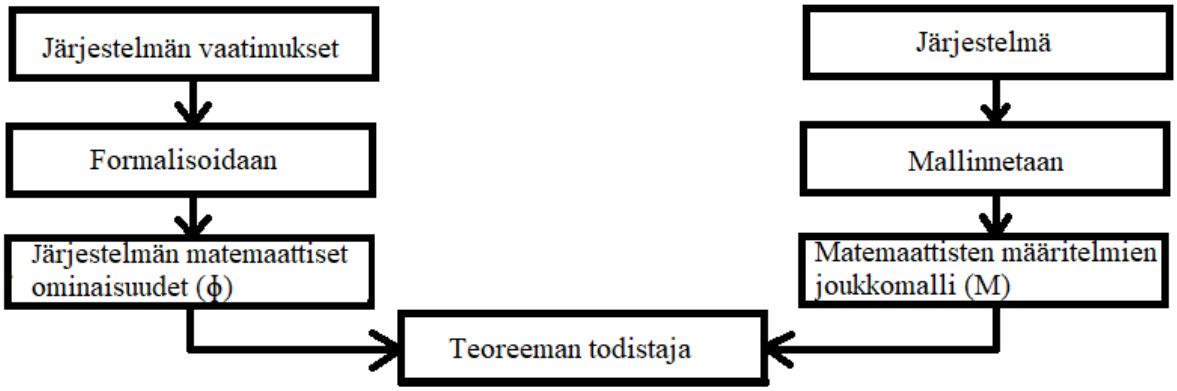
Kuva 2. Formaali Tarkastustekniikat [9]

Kuvassa 3. on esitetty seuraavat vaiheet, jotka selittävät mallin tarkastus prosessin, jossa järjestelmä mallinnetaan joukkona tiloja, joissa on joukko tilojen välisiä siirtymiä, jotka kuvaavat, kuinka järjestelmä siirtyy tilasta toiseen vasteena sisäisille tai ulkoisille ärsykkeille. Prosessissa luodaan formaalisesti järjestelmämallille M todennettava ominaisuus järjestelmän vaatimuksista käyttämällä ominaisuuden määrityskieltä, kuten SVA, jossa ominaisuus ϕ on kaava, joka on kuvaus suunnittelun käyttäytymisestä. Prosessi suorittaa tarkastuksen mallin tarkistajalla selvittääkseen, täyttääkö järjestelmän malli M kaavan ϕ, eli ovatko ne yhdenmukaiset. Jos malli ei täytä ominaisuutta, luodaan vastaesimerkki, jossa vastaesimerkki on ärsyke, joka rikkoo ominaisuutta. Vastaesimerkki auttaa löytämään simulaatiossa virheen sijainnin järjestelmämallissa. Mallin tarkastuksen varmennusprosessi on täysin automaattinen, kun järjestelmämalli ja ominaisuusspesifikaatio on toimitettu mallin tarkistajalle. Järjestelmän tulisi kuitenkin olla pieni mallin tarkistimen käsittelemien tilojen lukumäärän suhteen [9].



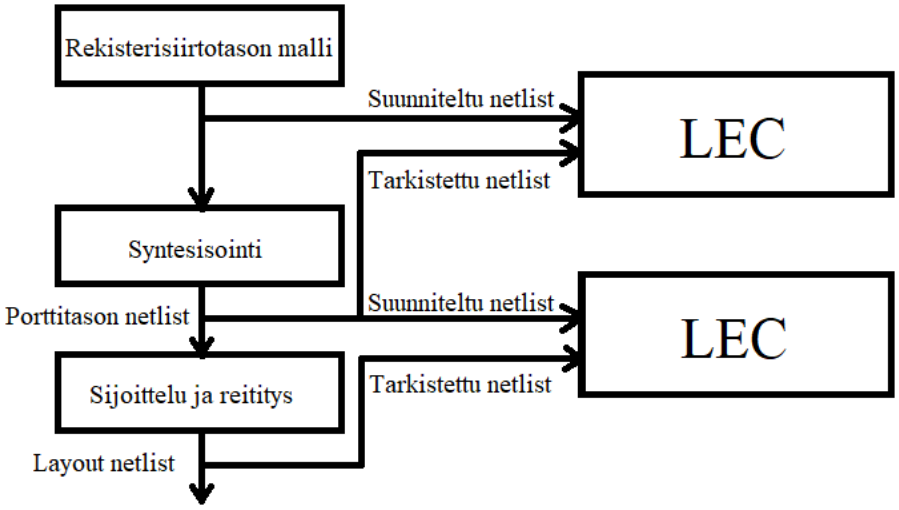
Kuva 3. Mallin tarkastuksen prosessi [9]

Kuvassa 4. on seuraavat vaiheet teoreeman todistamisen prosessissa, jossa mallinnetaan järjestelmä matemaattisten määritelmien joukkomallina M formaalisessa matemaattisessa logiikassa. Formalisoidaan suunnitelman vaatimuksista seuraavat järjestelmän matemaattiset ominaisuudet ϕ, jonka jälkeen käytetään teoreemaan todistajaa varmistamaan, että järjestelmä täyttää vaatimukset. Teoreeman todistamisen suurin etu on, että se pystyy käsittelemään hyvin monimutkaisia järjestelmiä, mutta se ei kuitenkaan ole täysin automaattinen. Se vaatii manuaalista apua todistusten suorittamiseen, mikä vaatii aikaa ja asiantuntemusta. Epäonnistuneen todistuksen tapauksessa ei myöskään luoda vastaesimerkkejä, mikä vaikeuttaa virheen paikantamista [9].



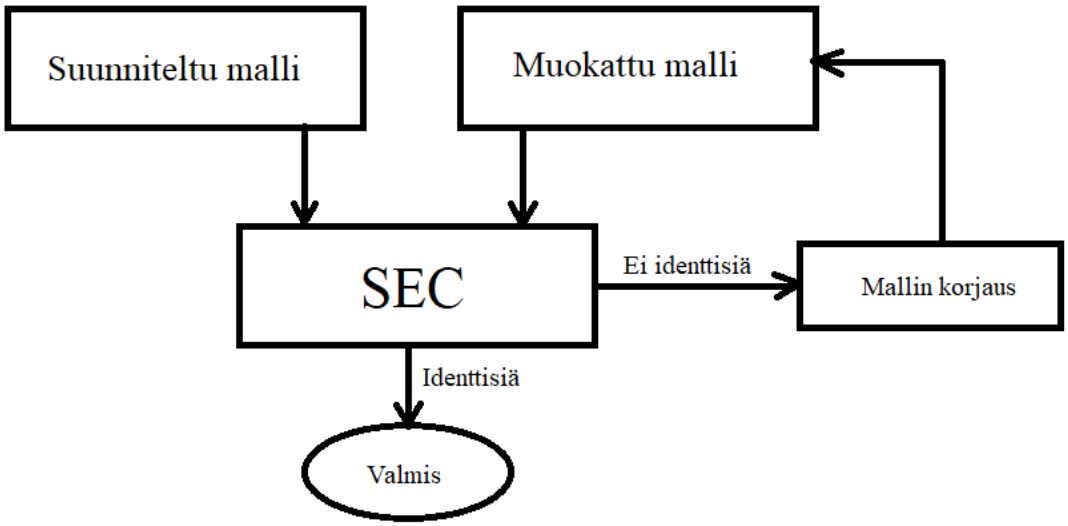
Kuva 4. Teoreeman todistamisen prosessi [9]

LEC on prosessi, jolla varmistetaan, että kahdella mallilla on sama yhdistelmälogiikka rekisterien välillä. Molemmissa verrattavissa olevissa malleissa tulisi myös olla sama määrä rekistereitä. Tätä tekniikkaa käytetään varmistamaan, että kaksi eri abstraktiotasolla olevaa mallia ovat toiminnallisesti identtisiä, esimerkiksi kuvassa 5., että porttitason netlist on toiminnallisesti sama kuin layout netlist [9].



Kuva 5. Loogisen vastaavuuden tarkastus [9]

SEC on prosessi, jolla varmistetaan, että kaksi mallia ovat toiminnallisesti identtisiä ja että ne antavat samat lähdöt, kun niillä on samat tulot. SEC vertaa kahden mallin peräkkäistä logiikkaa, joilla voi olla erilaisia toteutuksia. Tämä on monimutkainen prosessi, ja siksi se rajoittuu hyvin suunnittelun kokoon. Kun IC:n rakennetta muutetaan viime hetkellä sisällyttämällä siihen joitain toiminnallisia, ajoitus-, teho- tai muita korjauksia tai lisälogiikkaa, kuten skannauslogiikkaa tai tehonsäätöpiiriä. Tällaiset muutokset on tarkistettava. Vakiovarmennusmenettelyt vievät paljon aikaa ja lisäävät siten markkinoille tuloaikaa. Kuvassa 6. SEC vertaa muokattua mallia suunniteltuun malliin ja varmistaa, että ne ovat toiminnallisesti identtisiä [9].



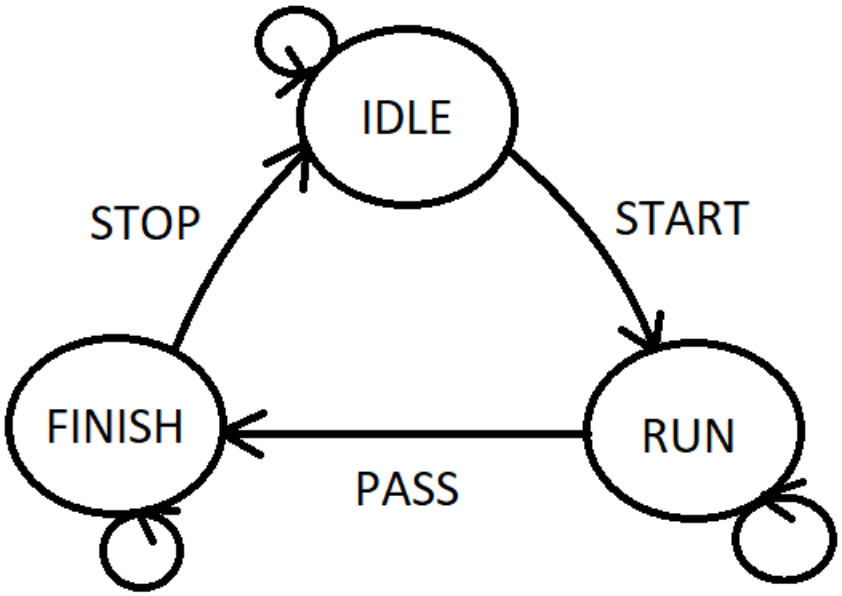
Kuva 6. Peräkkäisen vastaavuuden tarkastus [9]

2.4 Väitteiden formaali tarkastus

Väitteiden formaali tarkastus (Engl. Assertion-Based Verification, ABV) on automaattinen menetelmä, toisin sanoen sitä kutsutaan aikaisemmin kerrotuksi ominaisuuksien tarkistukseksi, jossa suunnitelmien tarkistamiseen käytetään ominaisuuksia, jotka on kirjoitettu käyttäen ominaisuuskieltä SVA. ABV sisältää suunnittelun analyysin, jossa on päätavoitteiden määrittäminen väitteille, ja asianmukaisten ominaisuuksien, väitteiden ja kattavuuden toteuttaminen, sekä tarkastus staattisella formaalisella tarkastuksella, dynaamisella simuloinnilla, tai näiden yhdistelmällä. ABV voidaan suorittaa seuraavien projektivaiheiden aikana, kuten tehtäessä suunnitelman spesifikaatiot, suunnittelun arkkitehtuuri ja toteutus, sekä todentamisympäristön suunnittelu ja toteutus. ABV käyttöä voidaan soveltaa olemassa olevaan malliin, jossa on tunnettuja ongelmia tai suunniteltuja johdannaisia, uuden mallin kehittämisen aikana, sekä järjestelmän yksittäisiin osiin tai koko projektiin. Esimerkiksi kuvassa 7. on tilakone, jossa on kolme tilaa IDLE, RUN ja FINISH. Tämä tilakone on koodattu SV-kielellä, ja se käyttää tilamuuttujaa STATE ja seuraavaa tilaa kuvaavaa tilamuuttujaa NEXT_STATE. Tilakone toimii niin, että seuraavaa tilaa ohjataan kellosta ja resetoinnista riippuen always_ff-proseduurin sisällä riveillä 8–14 kuvassa 8., sekä tilakoneen toiminnallisuus on always_comb -proseduurin sisällä riveillä 16–41 kuvassa 9. Tilakoneen oletettua käyttäytymistä kuvaamaan kirjoitetaan kolme väitettä kuvassa 10. Moduulissa on sisääntuloina: kello (CLK), resetointi

(RST_N), sekä toimintasignaalit (START, PASS, STOP). Assert-väitteet ominaisuuslauseiden sisällä kuvaavat tilakone mallin oletettua käyttäytymistä, että jos tilamuuttaja on IDLE-tilassa ja START-toiminto nähdään, niin seuraavan kellojakson jälkeen tilamuuttujan tilan pitäisi olla RUN. Vastaavasti, jos tila on RUN ja nähdään PASS-toiminto, niin seuraavalla kellojaksolla tilamuuttujan pitäisi olla FINISH. Lopuksi, jos tila on FINISH ja nähdään STOP-toiminto, niin tila palautetaan IDLE-tilaan seuraavalla kellojaksolla. Cover-väitteet ominaisuuslauseiden sisällä taas kirjaavat ylös, kuinka usein tietty ominaisuus toteutui. Väitteet eivät ole toiminnassa, kun resetointi on epätosi, sekä resetoinnissa tila palautuu aina IDLE-tilaan.

Kun väitteet ja RTL-koodi on kirjoitettu, ne tarkastetaan formaali tarkastustyökalua käyttäen, kuten OneSpin 360-DV Verify -ohjelmaa. Tarkastuksessa tarkastetaan, että väitteet joko toteutuvat tai eivät, jolloin tiedetään, toimiiko koodattu tilakone väitteiden mukaan. SVA sisältää kaikki mahdolliset käyttäytymiset suunnittelusta, joten jos kaikki väitteet toteutuvat, on kyseessä Holding SVA, joka tarkoittaa, että kaikki väitteet ovat formaalisesti ja tyhjentävästi tarkastettu. Kun taas epäonnistunut SVA tarkoittaa, että löydettiin vastaesimerkki, joka edustaa suunniteltua suunnittelukäyttäytymistä. Vastaesimerkin pystyy katsomaan esimerkiksi käyttämällä OneSpin 360 DV-ohjelmassa väitteen debuggeria [10].



Kuva 7. Tilakone

```
1 module statemachine_svatest
2   (input logic clk, rst_n, start, pass, stop);
3
4    typedef enum  logic [1:0]
5         { IDLE = 2'b00, RUN = 2'b01, FINISH= 2'b10 } statemachine_t;
6    statemachine_t state, next_state;
7
8   always_ff @(posedge clk or negedge rst_n)
9        begin : statemachine
10        if (!rst_n)
11         state <= IDLE;
12        else
13         state <= next_state;
14       end : statemachine
15
```

Kuva 8. Tilakoneen koodi riveiltä 1–15

```
16   always_comb
17        begin : statemachine_logic
18           case (state)
19            IDLE:
20             begin
21              if (start)
22                next_state = RUN;
23              else
24                next_state = IDLE;
25             end
26            RUN:
27             begin
28              if (pass)
29                next_state = FINISH;
30              else
31                next_state = RUN;
32             end
33            FINISH:
34             begin
35              if (stop)
36                next_state = IDLE;
37              else
38                next_state = FINISH;
39             end
40           endcase
41        end : statemachine_logic
42
```

Kuva 9. Tilakoneen koodi riveiltä 16–42

```
43   property r_follow_start;
44         @(posedge clk) disable if (!rst_n)
45          (state == IDLE) && start |=> (state == RUN);
46   endproperty;
47
48   ar_follow_start: assert property (r_follow_start)
49         else $error("Next state isn't RUN.")
50   cr_follow_start: cover property (r_follow_start)
51
52   property r_follow_pass;
53         @(posedge clk) disable if (!rst_n)
54          (state == RUN) && pass |=> (state == FINISH);
55   endproperty;
56
57   ar_follow_pass: assert property (r_follow_pass)
58         else $error("Next state isn't FINISH.")
59   cr_follow_pass: cover property (r_follow_pass)
60
61   property r_follow_stop;
62         @(posedge clk) disable if (!rst_n)
63          (state == FINISH) && stop |=> (state == IDLE);
64   endproperty;
65
66   ar_follow_stop: assert property (r_follow_stop)
67         else $error("Next state isn't IDLE.")
68   cr_follow_stop: cover property (r_follow_stop)
69 endmodule
```

Kuva 10. Väitteet tilakoneelle tilakoneen koodi riveiltä 43–69

3 AIHEEN TOTEUTUS

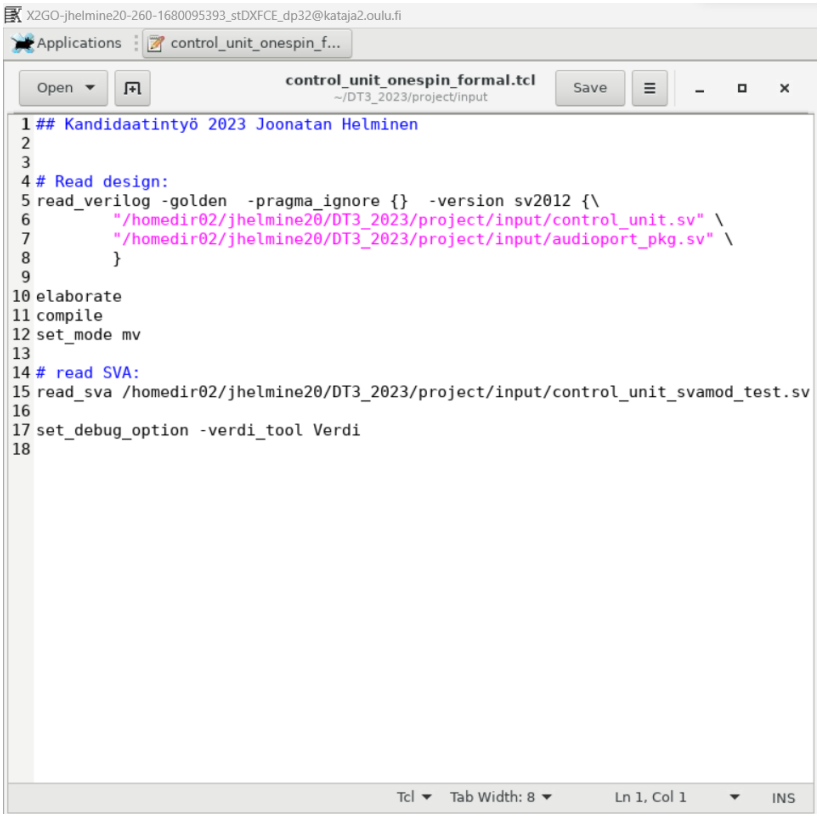
3.1 Toteutuksen sisältö

Tässä käytännönpuolentyön toteutuksessa käytetään Digitaalitekniikka 3 -kurssilla keväällä 2023 projektityöhön liittyen tekemiäni control_unit-moduuliin tehtyjä RTL-koodi-, SVA- sekä testitiedostoa apuna SV väitteiden formaalia tarkastusta varten OneSpin 360 Design Verification-ohjelmassa. Kurssilla toteutetun projektin aiheena oli I2S-äänilähdön rajapintalohko, johon tehtiin audioportin suunnittelu ja toteutus System-On-Chip (SOC) sovelluskohtaiselle integroidulle piirille. Audioportti toimii vastauslaitteena SOC:n oheisväylällä, joka perustuu AMBA Advanced Peripheral Bus (APB) -spesifikaatioon. Audioportti on kytketty SOC:n pääväylään väyläsillan kautta, joka tuottaa APB-protokollan mukaisia signaaleja APB-transpondereille SOC:n pääväyläsignaaleista ja multipleksit lukevat dataväyliä APB-transpondereista pääväylään. Audioportti on myös kytketty keskeytysohjainyksikköön signaalilla (irq_out). Control_unit-moduuli toimii audioportissa ohjausyksikkönä, joka toteuttaa projektissa APB-väylärajapinnan logiikan ja muistikartoitettuja rekisteripankkeja ohjaus-, konfigurointi- ja äänidatan tallentamista varten. Se myös dekoodaa ohjaussignaalin arvot komentorekisteriin kirjoitetusta tiedosta. Ohjausyksikkö sisältää myös logiikan datan suoratoistoon dsp_unit-moduulille ja keskeytysten generoimiseksi uuden audiodatan lataamisen äänipuskurirekistereihin. RTL-kooditiedostona toimii control_unit.sv, jonne on tehty suunniteltu ja tarkistettavana oleva ohjausyksikkö-moduuli. Väitteet on sijoitettu niin kutsuttuun ”Checker”-moduuliin, joka on tässä työssä erillisessä control_unit_svamod_test.sv-tiedostossa toisin sanoen SVA-tiedostossa. Ohjelmaa varten SVA-tiedostoon instantioidaan moduulit kuvan 11. lauseella [5].

```
bind control_unit control_unit_svamod control_unit_assertions_inst ( .*);
```

Kuva 11. Moduulien instantiointi-lause

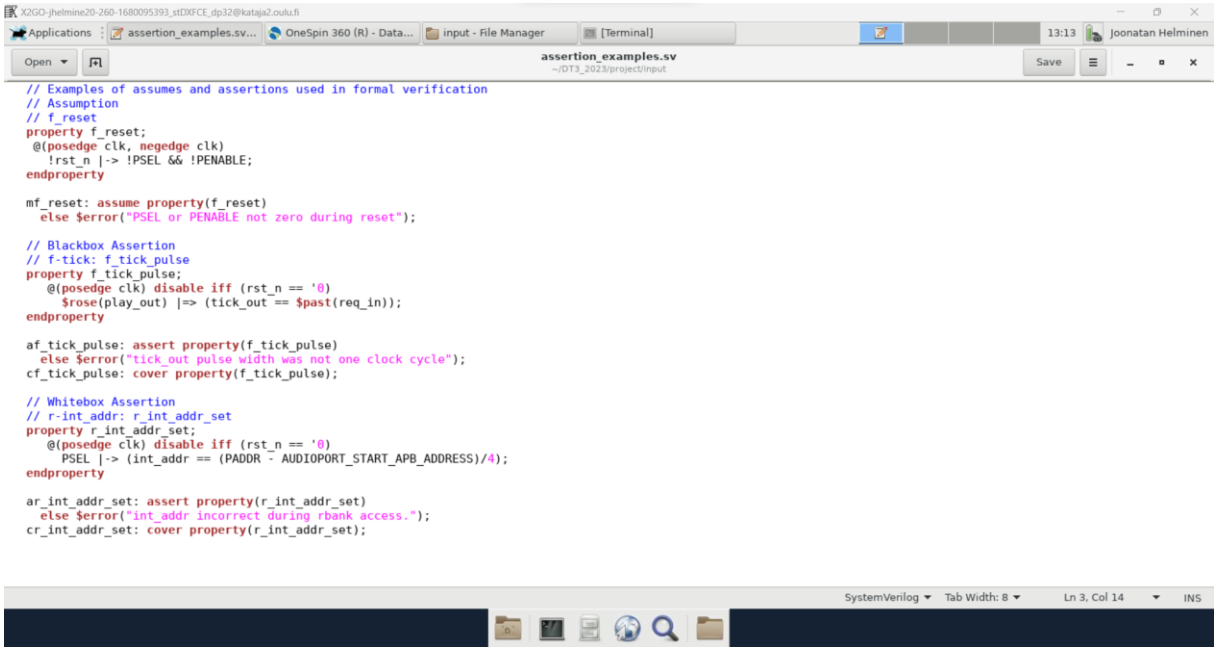
OneSpin 360 DV -ohjelma tarjoaa formaalisen RTL-moduulin toiminnan tarkistuksen, kun malli on laadittu ja koottu, jolloin voidaan mv-tilassa tarkistaa rakenteen johdonmukaisuus ja yhtäpitävyys suunnitelman kanssa käyttämällä automaattista tarkistusta apuna. Pystytään tarkistamaan matalan tason väitteet, kuten SVA, sekä tarkistamaan suunnittelun todellinen toimivuus odotettuun toimintaan verrattuna, mukaan lukien ominaisuuksien kehittäminen tai suunnittelun ja ominaisuuksien virheenkorjaus. Voidaan myös tarkistaa ominaisuusjoukon täydellisyys ja lisätä ominaisuuksia tarvittaessa. Kuvassa 12. on luomani control_unit_onespin_formal.tcl- testitiedosto, joka sisältää kaikki tiedostot, joita käytetään formaali tarkastuksessa OneSpin 360 DV-ohjelmassa. Testitiedosto kutsutaan ohjelman sisällä lukemaan lähdetiedostot, kuten RTL-kooditiedoston control_unit.sv ja siihen sisälletyn pakettitiedoston audioport_pkg.sv. Pakettitiedostoon on kirjoitettu projektin suunnitteluparametrit ja rekisteröity rekisteriosoitteet APB-osoiteavaruuksiin sekä sisäiseen osoitemuistirekisteriin nimeltä int_addr, joita käytetään RTL-moduulissa sekä väitemoduulissa. Lähdetiedostojen lukemisen jälkeen testitiedosto laatii ja kokoaa mallin ja siirtyy ohjelman mv-tilaan, jossa luetaan SVA-tiedosto sekä lisäksi kutsutaan virheenkorjausta varten Verdi debuggerityökalu toiseksi virheenkorjaustyökaluksi [10].



Kuva 12. Testitiedosto

3.2 Tarkistettavat väitteet

Tarkistettavina väitteinä on Control_unit_svamod_test.sv-tiedostossa assume-, whitebox- ja blackbox-väitteitä. Assume-väitteet kuvaavat ominaisuuksia RTL-koodissa, jotka oletetaan olevan aina totta. RTL-suunnittelua valvovia väitteitä kutsutaan whitebox-väitteiksi, koska ne voivat viitata suunnittelun sisäisiin muuttujiin, jotka määritellään RTL-suunnitteluvaiheessa. Whitebox-väitteitä käytetään tarkastuksessa sen varmentamiseksi, että RTL-koodari on toteuttanut RTL-suunnittelijan tarkoituksen oikein eli koodi ja suunnitelma ovat yhtäpitävästi toteutettu. Blackbox-väitteet tarkistavat toiminnallisten vaatimusten spesifikaatioissa määritellyt ominaisuudet, ja ne voivat viitata vain suunnittelun tuloihin ja lähtöihin. Assume- ja whitebox-väitteet on tehty projektin tekijän toimesta jo valmiiksi, mutta blackbox-väitteet kirjoitettiin itse käyttämällä apuna kurssilla control_unit-moduulin teknisissä tiedoissa kerrottuja kuvauksia väitteistä, jotka varmistavat tekemäni RTL-koodin toiminnan oikeellisuuden. Kuvassa 12. on esimerkki assume-, whitebox- ja blackbox-väitteestä, joita käytetään RTL-moduulin toiminnan oikeellisuuden tarkastuksessa. Assume-väite luo formaaliselle tarkastusohjelmalle oletusarvon eli rajoituksen, kun nollaus on epätosi, niin PSEL ja PENABLE ovat myös epätosia samalla kellojaksolla, silloin kun ohjelma todistaa väitteitä oikeiksi tai vääriksi. Blackbox-väite taas varmistaa sen, että jos play_out on tosi, seuraavan kellojakson tick_out on oltava sama kuin req_in tosi arvo, joka oli edellisellä kellojaksolla, jos play_out on edelleen tosi toisella kellojaksolla. Whitebox-väite varmistaa sen, että kun PSEL on tosi, niin samalla kellojaksolla int_addr-osoitemuistirekisterissä pitää olla kuvan 13. mukainen osoite [5].



Kuva 13. Esimerkki formaali tarkastettavista väitteistä

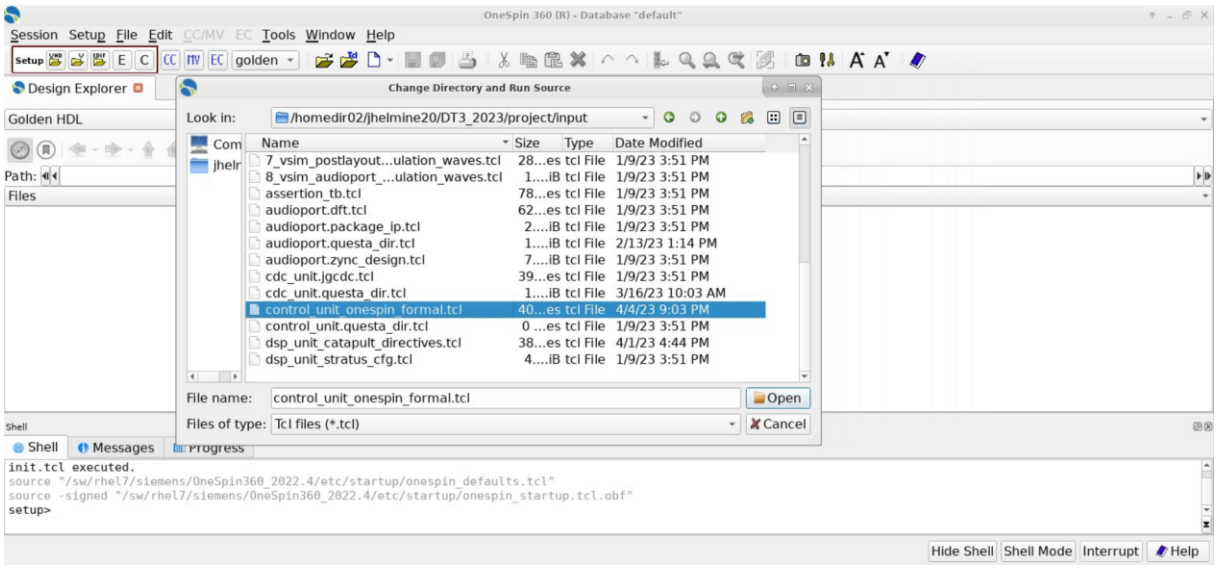
Väitteille tulee OneSpin 360 DV-ohjelman jokaisella tarkistusyrityksellä jokin seuraavista tuloksista [10]:

- Hold/pass tarkistustulos, jossa väitteen on todistettu toimivan kaikissa aktiiviset rajoitukset täyttävissä simulaatiojäljissä rajoittamattomalla syvyydellä.
- Fail(n) tarkistustulos, jossa väitteen tarkistus epäonnistui n jakson jälkeen nollauksesta. Virheenkorjausta varten tarjotaan vastaesimerkkinä jäljitys, joka alkaa nollauksesta.
- Hold_bounded(n) tarkistustulos, jossa nollauksen vastaesimerkkiä ei ole määritettyyn syvyyteen n asti. Ominaisuutta ei ole yleisessä tapauksessa todistettu. Saatavilla on vastaesimerkki, joka alkaa jostain mielivaltaisesta tilasta, joka ei ole nollattu, eikä sitä siksi voida saavuttaa.
- Vacuous/vacuous_bounded(n) tarkistustulos, joka osoittaa, ettei todistustuloksia eikä vastaesimerkkejä ole löydetty määritettyyn syvyyteen n asti.

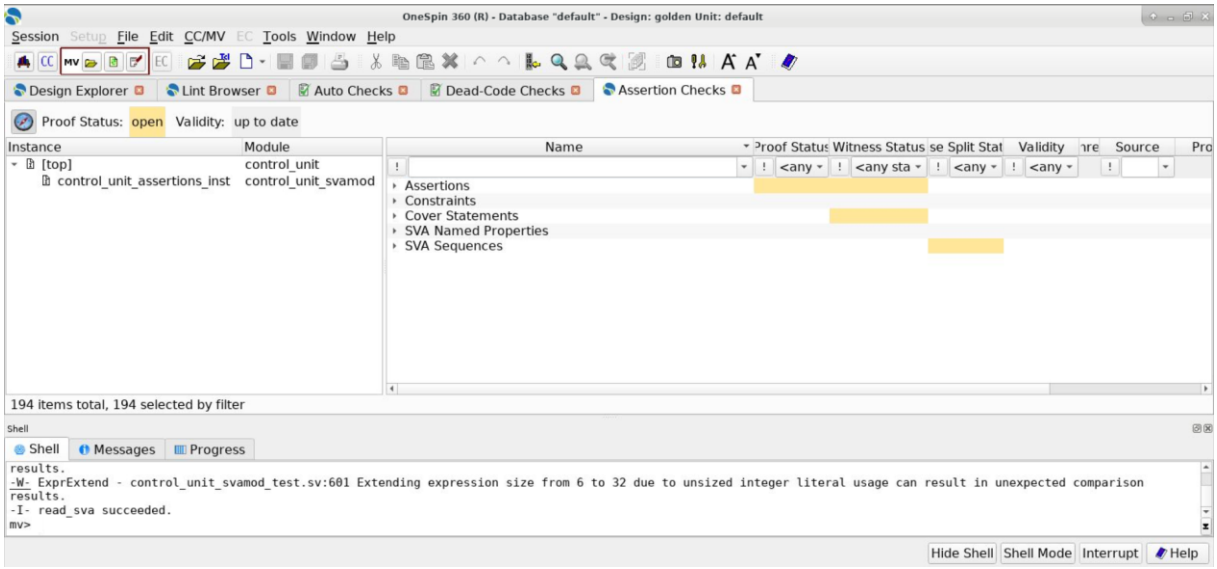
Mahdollisia syitä väitteiden epäonnistumiseen ovat esimerkiksi suunnitteluvirhe RTL-koodissa, väärinkirjoitettu ominaisuus tai puuttuvat ympäristörajoitukset, joita pystytään selvittämään vastaesimerkin avulla, joka pystytään luomaan käyttämällä apuna OneSpin 360- tai Verdi-debuggerityökalua. Vastaesimerkki on debuggerin tuottama diagnostiikkatieto, joka on yksi esimerkki siitä, kuinka suunnittelukäyttäytyminen voi rikkoa väitettä. Pitää huomioida, että ei ole varmuutta siitä, että sama vastaesimerkki saadaan uudelleen, jos komento suoritetaan uudelleen, koska yleisesti ottaen on olemassa useita mahdollisia suunnittelunkäyttäytymisen esimerkkejä, jotka voivat saada väitteen epäonnistumaan. Yksi vastaesimerkki valitaan muiden joukosta heuristisesti. Samaa vastaesimerkkiä voidaan kuitenkin käyttää eri debuggereissa niin kauan kuin tarkistuskomentoa ei suoriteta uudelleen [10].

3.3 OneSpin 360 DV-ohjelmalla väitteiden formaali tarkastus

OneSpin 360 DV-ohjelman asetukset saadaan Digitaalitekniikka 3-kurssin X2Go-etätyöpöytäympäristössä Linux-palvelimella terminaaliin komennolla “source /usr/local/contrib/eda_tools_setup.bash”, jonka jälkeen ohjelma lähtee käyntiin komennolla ”onespin”. Formaali tarkastettavat RTL-tiedostot saadaan auki ohjelman sisällä yllä olevassa rivivalikosta painamalla ”session”, ja valitsemalla ”cd -run source”, joka aukaisee kuvan 14. mukaisen valikkoikkunan, josta etsitään luotu testitiedosto nimeltä control_unit_onespin_formal.tcl. Testitiedoston aukaisemisen jälkeen ohjelma suorittaa sinne kirjoitetun koodin laatimalla ja kokoamalla lähdetiedostot ja lukemalla SVA-tiedoston, jolloin aukeaa kuvan 15. mukainen ”Assertion Checks”-ikkuna. Ikkunasta pystytään katsomaan, mitä kaikkia assert-väitteitä, rajoituksia eli assume-väitteitä, cover-väitteitä sekä SVA-sekvenssejä SVA-tiedostosta luettiin.

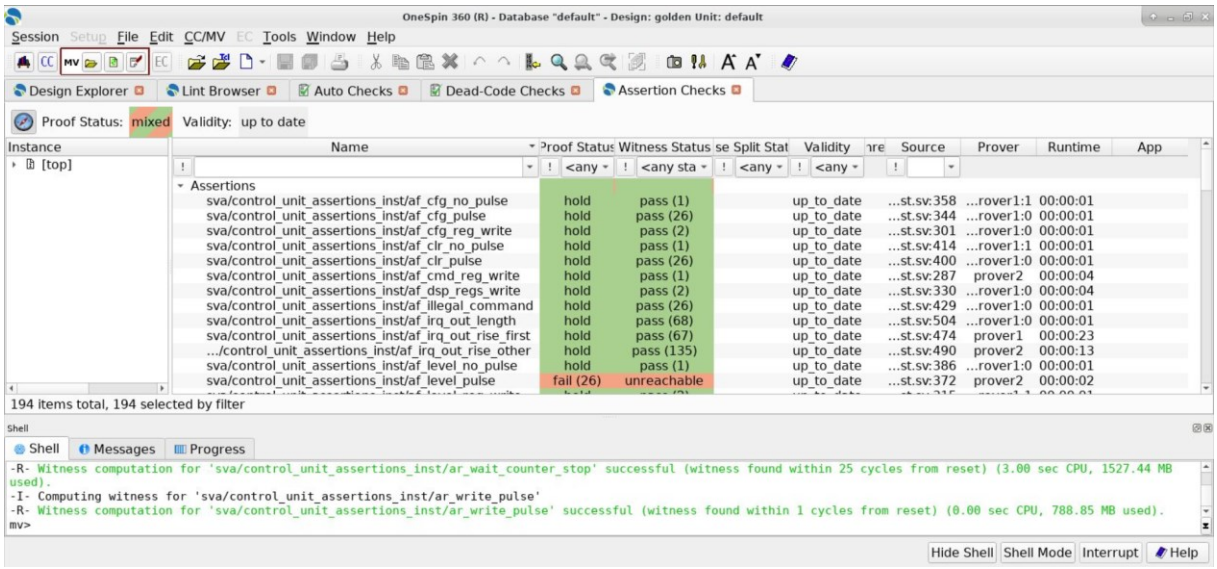


Kuva 14. Testitiedoston suoritusikkuna



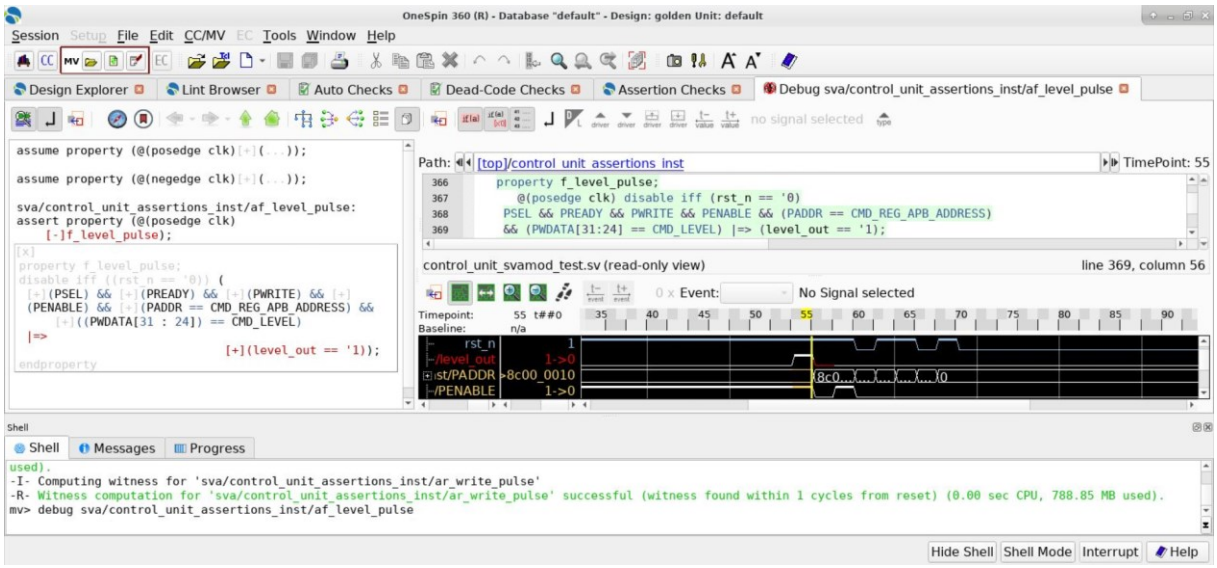
Kuva 15. Assertion Checks-ikkuna

Käyttäjä voi tarkistaa kaikki väitteet painamalla päämoduuli-tiedostoa hiiren kakkospainikkeella ja valitsemalla ”Check all”-painikkeen tai käyttämällä komentotulkissa mv-tilassa komentoa ”check -all [get_checks]”. Väitteiden formaali tarkastuksen suoriuduttua Assertion Checks-ikkunassa kuvassa 16. nähdään kaikkien väitteiden tulokset. Huomataan, että yksi väitteistä epäonnistuu, jolloin ohjelma on luonut vastaesimerkin sille.

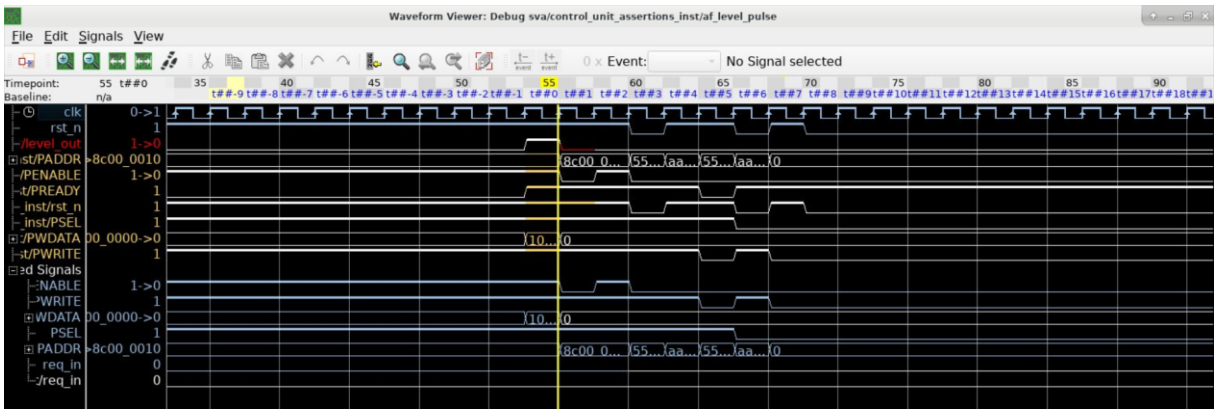


Kuva 16. Onespin 360 DV-ohjelman 1. tulokset

Käyttäjä pystyy myös tarkistamaan yksittäisiä väitteitä, kuten epäonnistuneen väitteen tapauksessa ohjelman Shell-ikkunassa mv-tilan ollessa käytössä komennolla ”check sva/ sva/control_unit_assertions_inst/af_level_pulse”, jonka jälkeen väitteen tulos näkyy samalla tavalla kuin kuvan 16. ”Assertion Checks”-ikkunalla, josta selviää, toteutuuko vai epäonnistuuko väite. Ohjelmalla osoitettiin, että väite af_level_pulse epäonnistuu tarkistaessa kaikki väitteet kerralla. Debuggerityökalulla pystytään katsomaan ohjelman luomaa vastaesimerkkiä graafisessa käyttöliittymässä. Debuggerityökalu suoritetaan joko napsauttamalla epäonnistunutta väitettä hiiren kakkospainikkeella ja valitsemalla "Debug Selection" tai käyttämällä seuraavaa komentoa Shell-ikkunassa mv-tilan ollessa käytössä ”debug sva/control_unit_assertions_inst/af_level_pulse”, jonka jälkeen aukeaa kuvan 17. debuggerityökalun-ikkuna, joka koostuu rakenteellisesta debuggerista vasemmalla ja suunnitteluvirheenkorjaus- ja aaltomuotojen-katseluohjelmasta oikealla. Rakenteellinen virheenkorjausohjelma antaa opastusta väitteen epäonnistumiseen johtamiseen todennäköisen syyn merkitsemällä tietyt väitteen ilmaukset punaisella. Lisäksi SVA-väitteen rakennetta voidaan tarkastaa eri hierarkioissa taittamalla ja avaamalla alilausekkeita. Tässä vastaesimerkissä SVA-operaattori (|=>) sekä (level_out == ’1) ovat merkitty punaisella, mikä osoittaa, että ne ovat vastuussa väitteen epäonnistumisesta. Aaltomuodonkatselu-ohjelma korostaa myös epäonnistumiseen liittyvät signaalit punaisella sekä sillä pystyy näkemään, miten kaikki signaalit käyttäytyvät vastaesimerkin tapauksessa ennen ja jälkeen keltaisen-kursoripystyviivan kohdalla kuvassa 18. Keltainen-kursoripystyviiva osoittaa ajan hetken, jolloin epäonnistuminen tapahtuu. Suunnitteluvirheenkorjaus-ohjelman avulla voidaan suoraan korjata väitteen rakennetta oikeaksi eli tässä vastaesimerkki tapauksessa korjataan SVA-operaattori (|->) samalla kellojaksolla tapahtuvaan, joka todettiin epäonnistumiseen johtavaksi syyksi.



Kuva 17. Debuggerin luoma vastaesimerkki

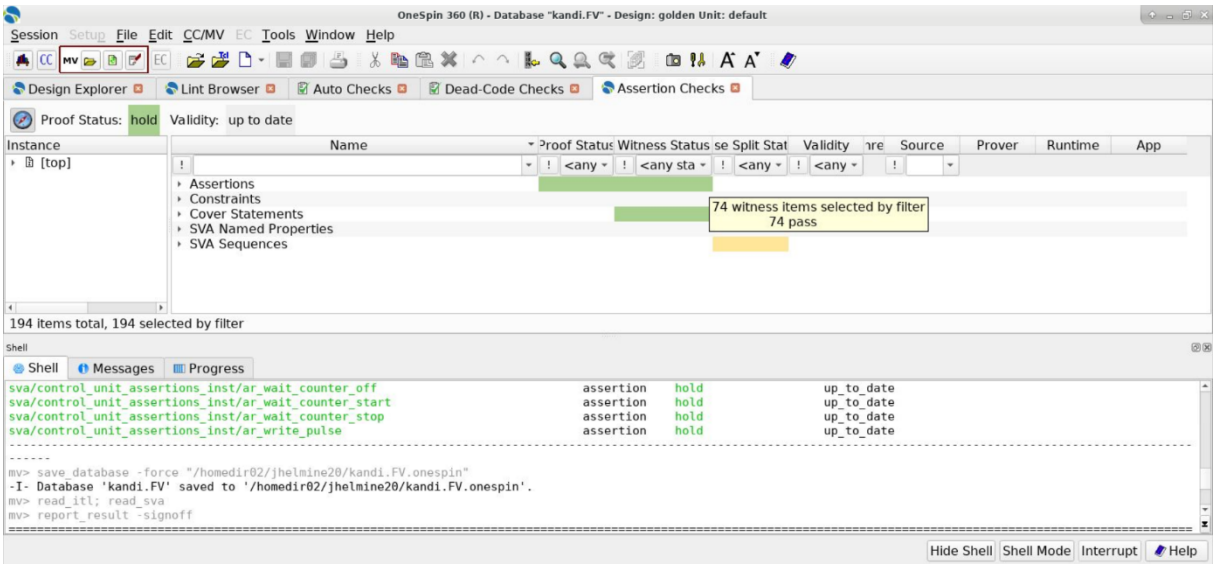


Kuva 18. Vastaesimerkin aaltomuodonkatselu-ohjelma

Kun väitteiden tarkastus on suoritettu uudelleen Onespin 360 DV-ohjelmalla, tulokset näkyvät kuvassa 19., jossa ”Proof Status: Hold” merkitsee sitä, että väitteiden tarkastus on toteutunut formaalisesti ja tyhjentävästi, eikä vastaesimerkkejä syntynyt. Keltainen tekstilaatikko tulee käyttäjälle esille hiiren siirtäessä ”Assertions”-listan kohdalle, joka kertoo, että 74 assert-väite on tarkistettu, sekä samalla tavalla cover-väitteiden kohdalla 47 suoriutuu. Väitteiden tulokset voidaan myös saada Shell-ikkunassa mv-tilan ollessa käytössä komennolla ”report_result -signoff”, joka tulostaa yhteenvedon väitteiden tarkastustuloksista, joiden tulokset ja määrät on koottu taulukkoon 2.

Taulukko 2. OneSpin 360 DV-ohjelman väitteiden tulokset

Assertion:	Result:	Count:
Assert	Hold	74
Assume	Pass	12
Cover	Pass	47



Kuva 19. Onespin 360 DV-ohjelman 2. tulokset

4 POHDINTA

Työssä onnistuttiin pääsemään tavoitteisiin, kuten tekemään käytännönpuolella SV-kielen väitteiden formaali tarkastus käyttämällä automaattista formaali tarkastustyökalu OneSpin 360 DV-ohjelmaa sekä opittiin syvemmin teorian pohjalta, miksi väitteiden ja formaali tarkastuksen käyttö on kannattavaa ja merkittävää laitteistojen oikeellisen toiminnan ja suunnittelun varmentamisessa. Väitteiden käyttö on kannattavaa, kun pystytään löytämään suunnitteluvirheet ja loogisesti väärät käyttäytymiset laitteistojen suunnittelussa, sekä formaalisen tarkastuksen tehokkuus ja oikeellisuus riippuvat suuresti väitteiden tarkkuudesta. Formaalisen tarkastuksen käytössä on merkittävää se, että pystytään jäljittämään vikoja, joita ei havaita pelkästään käyttämällä simulaatiota, kuten ”Corner Case”-tapaukset, sekä sillä pystytään osoittamaan, että suunnittelu täyttää sen vaatimukset ja paljastamaan suunnittelun epäselvyydet sekä suunnitteluvirheet käyttämällä väitteitä apuna.

Käytännön osuudella toteutettu formaali tarkastus OneSpin 360 DV-ohjelmalla antoi tyhjentävän tarkastustuloksen, jonka perusteella voidaan todeta, että suunnittelu täyttää väitteiden avulla tehdyt vaatimukset. Ohjelman käyttö myös opetti käyttämään debuggeria epäonnistuneiden väitteiden tapauksessa, kun ohjelma loi vastaesimerkin, jonka avulla pystyttiin selvittämään syy väitteen epäonnistumiselle. Ohjelmasta saadut tarkistustulokset kuvassa 19. ja taulukossa 2. ovat samat suoritetusti sekä määrällisesti verrattuna DT3-kurssilla käytetyillä ohjelmilla saatuihin tuloksiin.

Väitteiden formaali tarkistustulosten merkitys on kattava, vaikka ohjelmista saatujen tulosten välillä on pieniä eroavaisuuksia, jotka johtuvat ohjelmien sisäisistä toiminnoista. Tuloksien sekä helppokäyttöisyyden perusteella käyttökohteena kaikki ohjelmat, joita olen käyttänyt, ovat hyviä formaali tarkastustyökaluja perehtyessäni ohjelmien käyttöön. Väitteiden sekä formaalin tarkistamisen käyttö nopeuttaa ja helpottaa löytämään suunnitteluvirheet sekä epäselvyydet tarkistettavasta mallista, sekä auttavat varmentamaan mallin oikean käyttäytymisen toteutumisen. Jatkoa ajatellen pystyttäisiin tutkimaan ja perehtymään, miten eri ohjelmat formaali tarkistavat väitteet ohjelmien sisäisten toimintojen tapauksessa ja millaisia eroavaisuuksia sisäisissä toiminnoissa on.

5 YHTEENVETO

Tässä kandidaatintyössä toteutettiin SystemVerilog-laitteistokuvauskielessä esiintyvien väitteiden formaali tarkastus ensiksi perehtymällä aiheeseen teoriapuolella, jonka pohjalta tehtiin käytännönpuolentyö. Työn käytännön osuus tehtiin Digitaalitekniikka 3-kurssin X2Go-etätyöpöytäympäristössä Linux-palvelimella käyttämällä automaatista formaali tarkastustyökalu OneSpin 360 DV-ohjelmaa.

Teoriaosuudessa perehdyttiin SystemVerilog-kieleen, ja siinä esiintyviin väitteisiin, jotka ovat suuressa osassa laitteistojen verifiointia sekä validointia, eli vahvistamisprosessia, jossa väitteillä asetetaan kriteerit suunnittelutoteutuksella käyttämällä laitteistoa kuvaavia ominaisuuksia, sekä määrittämällä ja toteuttamalla laitteen käyttäytymismalli. Väitteillä pääasiassa pyritään tarkastamaan loogisesti väärät käyttäytymiset laitteistojen suunnittelussa simuloinneilla ja formaalisilla tarkastuksilla. Väitteitä käytetään formaalissa tarkastuksessa määrittämään formaalisilla menetelmillä todistettava ominaisuus sekä määrittelemään rajoitusympäristö kelvollisilla syötteillä DUT-toiminnalle. Formaalisen tarkistamisen tehokkuus ja oikeellisuus riippuvat suuresti näiden väitteiden tarkkuudesta.

Käytännön osuudessa tehtiin testitiedosto OneSpin 360 DV-ohjelmaa varten, jonka avulla saatiin ohjelman sisälle ennen formaalia tarkistamista luettua lähdetiedostot, kuten suunnitelman RTL-kooditiedosto sekä suunnitteluparametrejä sisältävä pakettitiedosto. Lähdetiedostoista laadittiin ja koottiin tarkistettava malli, jonka käyttäytymisen oikeellisuutta väitteillä sisältävällä SVA-tiedostolla formaali tarkastettiin ohjelmassa. Lähdetiedostojen sekä SVA-tiedoston sisällöt liittyvät Digitaalitekniikka 3-kurssin projektiin, jotka olin tehnyt valmiiksi aikaisemmin jo.

Tehdyn työn sekä ohjelmasta saatujen tulosten pohjalta voidaan todeta, että väitteiden ja formaalin tarkistamisen käyttö laitteistojen vahvistamisprosessissa nopeuttaa ja helpottaa suunnitteluvirheiden ja väärien käyttäytymismallien havaitsemista, sekä varmentaa ja vahvistaa mallin oikean käyttäytymisen toteutumisen.

6 LÄHDELUETTELO

[1] IEEE Std. 1800–2012, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language (2012) (Luettu 18.4.2023) URL: https://ieeexplore.ieee.org/document/6469140

[2] SystemVerilog Tutorial (2023) (Luettu 18.4.2023) URL: https://www.chipverify.com/systemverilog/systemverilog-tutorial

[3] Cerny, E., Dudani, S., Havlicek, J., & Korchemny, D. (2011) The Power of Assertions in System Verilog (Luettu 18.4.2023) URL: https://doi.org/10.1007/978-1-4419-6600-1

[4] SystemVerilog Assertions (2023) (Luettu 18.4.2023) URL: https://www.chipverify.com/systemverilog/systemverilog-assertions

[5] Digital Techniques 3 521406S-3004-Course (2023) (Luettu 18.4.2023) URL: https://moodle.oulu.fi/course/view.php?id=15805

[6] *SystemVerilog Immediate Assertions* (2023) (Luettu 18.4.2023) URL: https://www.chipverify.com/systemverilog/systemverilog-immediate-assertions

[7] *SystemVerilog Concurrent Assertions* (2023) (Luettu 18.4.2023) URL: https://www.chipverify.com/systemverilog/systemverilog-concurrent-assertions

[8] Seligman, E., Schubert, E. T., & Kumar, M. V. A. K. (2015). *Formal verification: an essential toolkit for modern VLSI design*. (Luettu 18.4.2023) URL: https://www.sciencedirect.com/book/9780128007273/formal-verification

[9] *Introduction to Formal Verification - EEWeb*. (Luettu 18.4.2023) URL: https://www.eeweb.com/introduction-to-formal-verification/

[10] OneSpin 360 DV-Verify Website (Luettu 18.4.2023) URL: https://onespin.com/products/360-dv-verify