Tekst: Hans van Thiel

Statische functionele verificatie

Een simulator is een vast onderdeel van elke EDA-suite die een HDL ondersteunt en zonder twijfel is simulatie de gangbare methode om aan het begin van de ontwerpcyclus fouten op te sporen. Formele verificatie begint echter langzamerhand een alternatief of in elk geval aanvulling te worden. Zo is statische functionele verificatie een variant die wordt geïmplementeerd in de tool Solidify van Averant. De Hardware Property Language van Solidify kan interactief worden toegepast, is gemakkelijk te leren en sluit aan bij de denkwijze van de elektronicaontwerper. Cisco Systems meldt goede resultaten bij het ontwerpen van een geheugencontroller.

De opkomst van Electronic Design Automation (EDA) heeft het mogelijk gemaakt steeds complexere systemen te ontwerpen. Met die toenemende complexiteit is echter ook de detectie van mogelijke ontwerpfouten een stuk ingewikkelder geworden. Computersimulatie is al tientallen jaren een goede en relatief goedkope methode, maar net als andere vormen van testen kan het alleen de aanwezigheid van fouten aantonen, niet de afwezigheid. Als de systemen ingewikkelder worden is niet alleen meer simulatie noodzakelijk - de kans op het missen van test vectoren die fouten aanwijzen neemt ook toe.

De toepassing van logisch-mathematische technieken om ontwerpfouten te detecteren krijgt dan ook, zowel in de academische wereld als bij de EDA-huizen, veel aandacht. Met deze formele verificatie is een ontwerp in principe volledig af te dekken tot op specificatieniveau. Ook kan equivalentie met een al bestaand (en goed werkend) systeem worden aangetoond.

Formele verificatie vereist beschrijvingen in formele talen en vervolgens tools die dergelijke beschrijvingen kunnen verwerken.

Men kan twee groepen onderscheiden, de *'model checkers'* die een toestandsruimte onderzoeken en de 'equivalentie checkers' die transformatieregels toepassen. Een zekere standaardisatie op dat gebied zou mooi zijn, maar daarvan is geen sprake. Zelfs de logisch- mathematische theorieën waarop tools zijn gebaseerd hoeven niet hetzelfde te zijn.

De elektronicaontwerper zal dus in de praktijk moeten vertrouwen op de correctheid van de tool. Wat dan belangrijk wordt is de werkbaarheid of gebruikersvriendelijkheid.

Hardwareverificatie is tenslotte geen vorm van wiskunde maar valt onder de elektronica.

Met Solidify kan de ontwerper condities formuleren, zogenoemde 'properties', waaraan het HDL-ontwerp (volgens de ontwerper) moet voldoen. De tool kan dergelijke 'properties' vervolgens automatisch verifiëren.

SFV

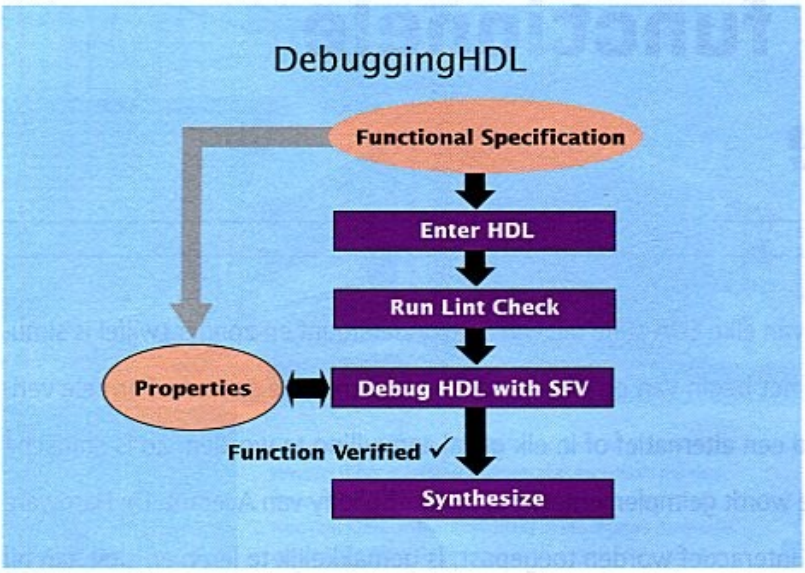
'Statische functionele verificatie' is de term die het bedrijf Averant gebruikt om de werking van haar tool 'Solidify' aan te duiden. De onderneming is in 1997 opgericht onder de naam HDAC en heeft een directie met een indrukwekkende staat van dienst in elektronicaontwerp en EDA. Averant richt zich uitsluitend op SFV en bedient klanten als Cisco, Compaq, Lucent en Hitachi.

De achterliggende methode wordt niet bekend gemaakt maar, gezien het verleden van president en oprichter Ramin Hojati, berust SFV waarschijnlijk op goed geoptimaliseerde model-checking. Het gebruik van 'properties' wijst ook in die richting.

Solidify kan blokken met zo'n 100K poorten verifiëren en schakelingen, zoals bus-arbiters, ALU's, geheugens, businterfaces, pijplijnen en toestandsmachines. De tool is geschreven in Java en heeft een IDE (Integrated Development Environment) voor het ontwerpen en debuggen van RTL-ontwerpen. Het gaat dan om schakelingen die geschreven zijn in Verilog of VHDL.

De plaats van SFV in de ontwerpcyclus staat aangegeven in figuur 1. Aan de hand van een functionele specificatie (die eventueel gewoon uit tekst kan bestaan) schrijft de ontwerper de 'properties' die hij of zij wil verifiëren. Solidify meldt of de regels gelden voor de ingevoerde HDL-beschrijving en zo nee, in welke gevallen niet. De ontwerper kan fouten in de HDL-bron herstellen en vervolgens opnieuw verifiëren.

Als een ontwerp op een groep eigenschappen is onderzocht dan kunnen, indien gewenst, weer nieuwe regels worden toegevoegd. SFV is in dat opzicht zowel interactief als schaalbaar. De tool heeft ook nog een mogelijkheid om te onderzoeken of alle waarden van alle signalen wel in een of andere 'property' aan de orde zijn gekomen. Maar deze 'Static Coverage Analysis' geeft geen garantie dat alle systeemfunctionaliteit in de opgestelde regels is afgedekt, alleen dat geen HDL-signalen zijn overgeslagen.



Statische Functionele Verificatie werkt op een RTL beschrijving in Verilog of VHDL. De ontwerper schrijft 'properties' aan de hand van de functionele specificatie en, mogelijkerwijs, voorgaande ontwerp iteraties. Het verdient aanbeveling de HDL beschrijving voor de SFV stap eerst nog op syntactische en semantische consistentie te testen met een zgn. 'Lint Checker'. (Bron: Averant)

HPL

Vanuit het standpunt van de ontwerper is de Hardware Property Language het belangrijkst. In deze taal, die veel weg heeft van Verilog, VHDL en ook C, kan hij of zij de condities formuleren. Neem bijvoorbeeld het Verilog ontwerp uit Listing 1. Het is een 8-bits teller die afhankelijk van een signaal *ud* omhoog of omlaag telt, mits signaal en hoog staat. Verder kan de teller gereset worden en kan invoer *cin* geladen worden op signaal *ld*. De eigenschappen (in HPL) die een ontwerper voor deze schakeling zou kunnen formuleren staan in Listing 2.
Men kan zich voorstellen dat de ontwerper allereerst de normale condities zal willen definiëren. Dat kan met de regel:

```
#define normal { !rst && !ld }
```

Hier staat: de identifier *normal* is de formule: reset signaal *rst is nul* en load signaal *ld is nul*. Deze identifier kan vervolgens gebruikt worden om functionaliteit te verifiëren. Zo wordt omhoog en omlaag tellen vastgelegd in de regels:

```
ud && en && normal => 'X(cnt) == cnt + 1;
!ud && en && normal => 'X(cnt) == cnt - 1;
```

Hierin staat => voor als...dan en 'X(...) voor de waarde van de term tussen haakjes op de volgende klokcyclus. De eerste regel betekent dus: als *ud*, *en* en *normal* hoog staan dan is de waarde van *cnt* op de volgende klokcyclus gelijk aan *cnt* plus 1. De tweede regel stelt dat met *ud* laag, *en* en *normal* hoog de waarde van *cnt* op de volgende klokcyclus met een verminderd moet worden. Solidify verifieert dit automatisch in ongeveer een seconde.
Static Coverage Analysis kan echter nu vaststellen dat de verificatie onvolledig is. Zo is er onder meer geen 'property' geschreven voor een laag enable signaal. De volgende regel corrigeert dit.

```
(normal && !en) => ('X(cnt) == cnt);
```

Als Inormal waar is en *en* op *nul* staat dan verandert de waarde van *cnt* niet op de volgende klokcyclus. Deze conditie is essentieel voor het correct functioneren van de teller.
Voor SFV maakt het geen verschil of er uitgangs- of ingangssignalen in de term vóór het implicatieteken staan. Een dergelijke omkering van eigenschappen ('reverse properties') is met simulatie niet mogelijk.
In Listing 2 staat de volledige opsomming van alle regels voor de teller uit Listing 1. Niet alle

```
module ud_counter(clk, ud, rst, ld, en, cin, cnt);
input clk, ud, rst, ld, en;
input [7:0] cin;
output [7:0] cnt;
reg [7:0] cnt, nxt_cnt;
always @(cnt or ud) begin
nxt_cnt[0] = !cnt[0];
nxt_cnt[1] = (ud ? cnt[0] : !cnt[0]) ^ cnt[1];
nxt_cnt[2] = (ud ? &(cnt[1:0]) : ~|(cnt[1:0])) ^ cnt[2];
nxt_cnt[3] = (ud ? &(cnt[2:0]) : ~|(cnt[2:0])) ^ cnt[3];
nxt_cnt[4] = (ud ? &(cnt[3:0]) : ~|(cnt[3:0])) ^ cnt[4];
nxt_cnt[5] = (ud ? &(cnt[4:0]) : ~|(cnt[4:0])) ^ cnt[5];
nxt_cnt[6] = (ud ? &(cnt[5:0]) : ~|(cnt[5:0])) ^ cnt[6];
nxt_cnt[7] = (ud ? &(cnt[6:0]) : ~|(cnt[6:0])) ^ cnt[7];
end
always @ (posedge clk) begin
if (rst) cnt <= 8 'b0;
else if (ld) cnt <= cin;
else if (en) cnt <= nxt_cnt;
end
endmodule
```

Listing 1. De Verilog code voor een Up/down Counter. De module heeft een 8-bits invoer cin en uitvoer cnt. Signaal ud bepaalt of de teller omhoog of omlaag gaat op kloksignaal clk (mits enable signaal en hoog is). Reset signaal rst zet de teller op nul en met behulp van load signaal ld kan de teller geladen worden met cin. Twee 8-bit registers cnt en nxt_cnt completeren de schakeling. (Bron: Averant)

```
// Synchroon laden
!rst && ld => 'X(cnt) == cin;

// Synchrone reset
rst => ('X(cnt) == 0);

// Definitie van normale operatie van de teller
#define normal { !rst && !ld }

// Op en neer tellen
ud && en && normal => 'X(cnt) == cnt + 1;
!ud && en && normal => 'X(cnt) == cnt - 1;

// De teller blijft gelijk
(normal &&!en) =>('X(cnt)==cnt);
('X(cnt) == cnt) => (normal && !en) || (rst && cnt == 0) ||
(!rst &&ld &&cnt==cin);

// De teller gaat omhoog
('X(cnt) == cnt+1) => (normal && en && ud) || (rst && (cnt==255)) ||
(!rst &&ld &&(cin==cnt+1));

// De teller gaat omlaag
('X(cnt)==cnt-1)=>((normal &&en &&!ud)||(rst &&(cnt==1))||
(!rst && ld && (cin==cnt-1)));
```

Listing 2. Verificatie van de teller uit Listing 1 in de Solidify Hardware Property Language (HPL). De 'X(...) operatie verwijst naar de waarde van de term tussen haakjes op de volgende klokcyclus. Zo betekent bovenstaande regel voor synchroon laden: als rst laag is en ld hoog dan moet de waarde van cnt op de volgende klokcyclus gelijk zijn aan cin. HPL heeft vergelijkbare operatoren om meer klokcyclussen te beschrijven. De taal heeft lexicaal en syntactisch overeenkomsten met Verilog, VHDL en C. (Bron: Averant)

beschikbare HPL-termen zijn in dit voorbeeld opgetreden; er zijn bijvoorbeeld ook expressies om meer dan een klokcyclus vooruit aan te geven.

Geheugencontroller

Er zijn wereldwijd ongeveer tien bedrijven die formele verificatietools op de markt brengen. Daarnaast zijn er academische tools beschikbaar en waarschijnlijk hebben grote halfgeleiderfabrikanten zelf hulpmiddelen ontwikkeld voor eigen gebruik.

Averant positioneert haar tool Solidify met name als alternatief voor simulatie en vooralsnog in het segment van middelgrote HDL blokken. Het grote voordeel ten opzichte van simulatie is dan de volledige afdekking inclusief 'reverse properties'. Statische Functionele Verificatie garandeert de correctheid van alle geformuleerde eigenschappen voor alle mogelijke waardencombinaties.

Op de System-on-Chip Design Conference in 2000 presenteerden twee ontwerpers van Cisco Systems een toepassing van SFV bij het ontwerp van een geheugencontroller. De basisarchitectuur omvatte de volgende blokken: Adress Decode Module, Memory Core Module, Pipeline Control Module en Register Module. Deze schakeling met 130 signaalpennen en 2350 flip-flops moest binnen 6 maanden worden geverifieerd. De ontwerpers melden dat verificatie van elk van de door hen geschreven eigenschappen niet meer dan een seconde kostte, wat neerkwam op een tijdsspanne van 1 - 2 minuten voor een module.

Volledige simulatie was hier niet haalbaar omdat het aantal in- en uitvoersignalen voor elke module eenvoudigweg te groot was. Wel deden de ontwerpers partiële simulaties als extra zekerheid en om het systeem als geheel te verifiëren. De conclusie was dat SFV een zeer nuttig instrument was gebleken.

Met dank aan Field Application Engineer G. Veurman van Prime EDA voor haar medewerking bij de totstandkoming van dit artikel. Eventuele fouten blijven uiteraard voor verantwoordelijkheid van de auteur.

Bronnen:
'Verifying a Simple Datapath', Solidify Application Note, Averant
M. Ross, S. Sambandan. 'Using Static Functional Verification in the Design of a Memory Controller', 2000 System-on-Chip Design Conference, Cisco Systems

Inl.: Prime EDA BV, Keesomstraat 17, 6717 AH Ede,
tel.: (0318) 692720,
web: www.prime-eda.com, www.averant.com