Verifica

Graziano Pravadelli
Dipartimento di Informatica Università di Verona

Agenda

- Verifica funzionale
  - Verifica dinamica
  - Verifica statica
  - Verifica semi-formale
  - Verifica basata su asserzioni
- Logiche temporali
- Linguaggi di verifica

Verifica funzionale

Does the proposed design do what is intended?

- Obiettivo:
  - Confermare che l'intento del progettista è stato rispettato durante l'implementazione
    - identificare errori (di progettazione)
- Richiede:
  - Specifica
  - Implementazione

Tipi di verifica funzionale

- Verifica dinamica
  - Per simulare il comportamento del design tramite l'uso di testbench e scenari
    - Simulazione logica e di guasto
    - Emulazione
- Verifica statica (formale)
  - Per provare matematicamente che il design soddisfa specifiche formali
    - Theorem proving
    - Equivalence checking
    - Model checking
- Verifica semi-formale
  - Verifica basata su asserzioni (ABV)

Verifica dinamica

- E' basata sulla generazione di un elevato numero di stimoli che vengono simulati per osservare il comportamento del sistema alle uscite
- Necessità di:
  - Modello del design in uno HDL
  - Simulatore per lo HDL
  - Testbench
  - Metodo per determinare la correttezza rispetto ai risultati della simulazione

Verifica dinamica

- Ottima per trovare errori ma…
- … non è in grado di dimostrare l'assenza di errori perché manca di esaustività
- Qualità della verifica misurata in termini di
  - Code coverage
  - Fault coverage

Simulazione logica

- Simula il comportamento del design
  - Semplice, veloce e scalabile …
  - … ma incompleta
- Necessita di metriche (*code coverage*) per capire quando fermarsi
  - Line coverage
  - Branch coverage
  - Condition converage
  - …

Simulazione di guasto

- Consiste nel simulare il design in presenza di un guasto
  - Comparando le uscite del design con guasto e quelle senza guasto riusciamo a capire se il guasto è testato o no
- Guasto non testato?
  - Verifica incompleta (testbench scarso)
  - Errore di progettazione (codice morto)
- Stessi pro e contro della simulazione logica, ma fornisce/usa *fault coverage* invece di code coverage

Emulazione

- Costruzione di una versione del sistema usando logica programmabile (FPGA)
- Utilizzo di tecniche di verifica dinamica sull'emulatore
- Permette di simulare molti testbench in poco tempo e di analizzare un elevato numero di scenari
- Richiede sintesi del modello ed è molto costosa

Verifica statica

- Usa metodi matematici per dimostrare la correttezza di sistemi
- Necessita di
  - Modello matematico del sistema composto da
    - Un insieme di stati
    - Una relazione di transizione tra stati
  - Linguaggio per specificare le proprietà da verificare
  - Metodo per dimostrare che il modello soddisfa le proprietà
- Dimostrazioni: manuali, semi-automatiche oppure completamente automatiche

Verifica statica

- Perché verifica statica?
  - Tecniche basate su simulazione non esaustive
  - matematicamente rigorosa
- ma
  - richiede personale altamente qualificato
  - problemi NP-completi → tempi di verifica lunghi
  - risorse spazio-temporali non sempre accettabili → non sempre si ha risposta
  - le specifiche possono essere incomplete

Theorem proving

- Dimostratore semi-automatico di teoremi
  - Mostra che una *congettura* è conseguenza logica di una insieme di *ipotesi* e *assiomi*
- Usa regole di inferenza
- Necessario essere estremamente esperti
- Processo interattivo

Equivalence checking

- Per dimostrare che due implementazione di un design sono equivalenti
  - Le implementazioni sono generalmente a differenti livelli di astrazione (es. RTL – gate), ma non necessariamente
  - TLM vs. RTL equivalence checking è un problema aperto estremamente interessante

Model Checking

- Introdotto da Clarke-Emerson e Queille-Sifakis (1981)
- Permette di verificare la *validità di un insieme di proprietà* specificate usando logica temporale *rispetto ad un modello*
- Se una proprietà non è valida fornisce un contro-esempio

Model Checking

- Come verifica una proprietà?
  - Clarke-Emerson
    - Costruisce un grafo degli stati completo del sistema
    - Propaga la formula da verificare nel grafo fino all'ottenimento di un punto fisso
  - Symbolic Model Checking
    - Utilizza OBDD
    - Limita esplosione degli stati

Verifica semi-formale

- Tenta di preservare vantaggi sia della verifica dinamica che della verifica statica rimuovendo i rispettivi svantaggi
- Approccio principale
  - Verifica basata su asserzioni

Verifica basata su asserzioni (ABV)

- Metodologia
  - basata su asserzioni
  - non ambigua
  - condivisibile tra vari approcci di verifica
  - per catturare l'intento (specifica) del progettista tramite
    - simulazione e/o
    - verifica formale
  - e verificare la corretta implementazione di tale intento

Specifica

- Spesso in linguaggio naturale, che non è automatizzabile
- Ambigua
- Difficile da condividere tra diversi processi di verifica (simulazione, verifica formale, emulazione, …)
- Difficile da formalizzare

Asserzione

- Affermazione di intento che può essere usata *per specificare in modo preciso e non ambiguo* il comportamento di un design
- Può specificare
  - Comportamenti interni
    - Es. struttura di una FIFO
  - Comportamenti esterni
    - Es. protocollo di un interfaccia

Asserzione

- Aspetto chiave
  - Permette di specificare ad alto livello *che cosa* si suppone debba essere fatto dal design senza dover specificare i dettagli di *come* debba essere implementato
    - *Astrazione* fondamentale per i processi di verifica

Controllabilità e osservabilità

- Fondamentali per ABV
- Controllabilità
  - Capacità di influenzare "qualcosa" nel design stimolando gli ingressi
- Osservabilità
  - Capacità di osservare sulle uscite gli effetti prodotti da "qualcosa" nel design

Testbench

- Ambiente virtuale usato per verificare la correttezza di un modello
  - Ha bassa controllabilità e osservabilità perché non analizza le strutture interne del design
- Per identificare un errore di progettazione deve soddisfare le seguenti condizioni:
  - Deve generare stimoli adeguati per attivare un errore
  - Deve generare stimoli adeguati per propagare l'effetto dell'errore sulle uscite

Asserzioni e osservabilità

- L'inserimento di asserzioni nel modello aumenta l'osservabilità
  - L'ambiente di verifica non dipende dalla capacità degli stimoli di propagare l'errore
  - Un comportamento errato o inaspettato viene catturato dalla asserzione vicino alla sorgente del problema (temporalmente e spazialmente)

Asserzioni e controllabilità

- Le asserzioni da sole non migliorano la controllabilità
  - Dobbiamo adottare ambienti di verifica che sfruttano adeguatamente le asserzioni (constraint-driven simulation) o utilizzare tecniche di model checking

Asserzioni dichiarative

- Sempre attive
- Concorrenti con altre componenti del design
- Adatte a specificare il comportamento delle interfacce a livello di blocco o di sistema

Asserzioni procedurali

- Eseguite sequenzialmente all'interno del codice
- Esprimono proprietà implementative all'interno del codice

Asserzioni e proprietà

- Il termine asserzione viene usato generalmente in ambito dinamico o semi-formale
- Il termine proprietà o formula viene usato generalmente in ambito formale
- Sono sostanzialmente intercambiabili

Proprietà desiderabili

- Liveness
  - Esprime l'eventualità che prima o poi si verifichi una condizione "buona"
  - Per rappresentare: terminazione, garanzia di servizio, starvation, …
  - Es.: Se P viene eseguito infinitamente spesso allora prima o poi deve produrre un output
- Safety
  - Esprime la necessità che non si verifichi mai una condizione "cattiva"
  - Per rappresentare: mutual exclusion, deadlock, FCFS, …
  - Es.: Non devo ottenere un output prima di aver fornito un input

Proprietà desiderabili

- Secondo Alpern e Schneider ogni proprietà è una intersezione di una proprietà di safety e una di liveness
  - Es.: L'evento E2 può eventualmente verificarsi e tutti gli eventi precedenti sono di tipi E1
- Se una proprietà di safety è falsa posso sempre mostrarlo con una traccia finita
  - Appena entro in uno stato in cui si verifica la condizione cattiva, la proprietà diventa falsa
- Se una proprietà di liveness è falsa posso mostrarlo solo con una traccia infinita
  - "Finchè c'è vita c'è speranza" (Cicerone)

Sistemi Reattivi

- Sistemi digitali sono sistemi reattivi (Pnueli 1986) in quanto:
  - Interagiscono continuamente con l'ambiente
  - Sono sistemi concorrenti: ogni porta logica calcola gli output in funzione degli input simultaneamente alle altre
- Per verificare sistemi reattivi servono logiche temporali

Logiche Temporali

- Logiche classiche sono in grado di descrivere solo situazioni statiche
- Logiche temporali
  - Estendono la logica proposizionale per descrivere sistemi reattivi
  - Utilizzano operatori temporali
    - Non solo pre e post condizioni ma anche …
    - … proprietà che descrivono cambiamenti nel tempo
  - Modellano il tempo in modo diverso:
    - Linear time
    - Branching time
    - … …

Logiche Temporali

- La semantica è definita usando il concetto di struttura di Kripke:
  - M = (S, R, L) dove:
    - S è l'insieme degli stati in cui il sistema evolve
    - R è la relazione di stato prossimo
    - L è una funzione che dato uno stato fornisce l'insieme delle proposizioni atomiche vere in tale stato
  - La verità di una formula è riferita allo stato presente

Logiche Temporali

- Caratteristiche generali:
  - R è una relazione di ordine parziale (antisimmetria, transitività)
  - Operatori
    - logica proposizionale: $\wedge$, $\vee$, $\neg$, $\rightarrow$
    - G: Gp è vera al tempo presente se p è vera sempre nel futuro
    - F: Fp è vera al tempo presente se p è vera prima o poi nel futuro

Linear Time Logic

- R è una relazione di ordine totale (dati due stati s, t è vero che s<t o s=t o s>t)
- Operatori
  - logica proposizionale
  - G, F
  - U: pUq è vera al tempo presente se p è vera sempre finchè non diventa vera q

Discrete Time Logic

- Ogni stato ha un successore immediato ed un predecessore immediato
- Operatori
  - Logica proposizionale
  - G,F,U
  - X: Xp è vera nello stato presente se p è vera nel prossimo stato
  - Xp è equivalente a falseUp

Branching Time Logic

- Ogni istante ha un passato unico, ma futuro indeterminato
  - Dati tre stati s, t, u, se t<s e u<s allora t<u oppure t=u oppure t>u
  - Considera tutti i possibili cammini che partono da uno stato
- Adatta per definire semantica di programmi non deterministici

Branching Time Logic

- Operatori:
  - Logica proposizionale
  - G,F,U,X
  - A: Ap è vera se per tutti i cammini vale p (necessità)
  - E: Ep è vera se esiste almeno un cammino in cui vale p (eventualità)

  - Esempi:
    - AF termina(prog) necessariamente prog termina
    - EF termina(prog) eventualmente prog termina

Computation Tree Logic

- E' un sottoinsieme della branching time logic definita da Clarke ed Emerson

- Operatori temporali occorrono solo in coppia
  - A oppure E seguiti da F, G, U oppure X

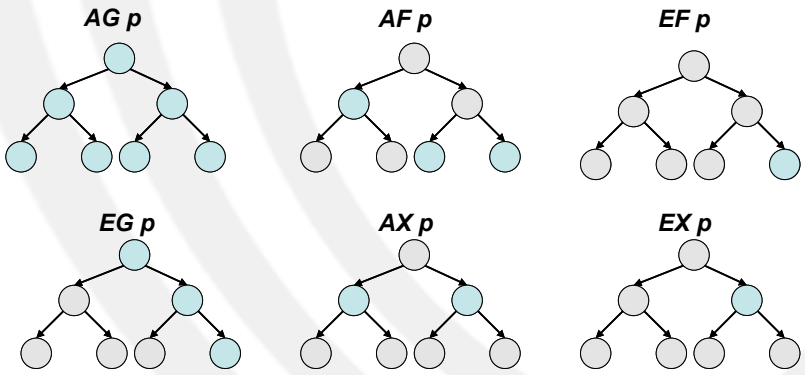
Computation Tree Logic

- Sintassi:
  - Ogni proposizione atomica è una CTL formula
  - Se f e g sono CTL formule allora lo sono anche: (f ∧ g), ¬ f, AX f, EX f, A (f U g), E (f U g)
  - Dalle precedenti derivano:
    - (f ∨ g) = ¬ (¬ f ∧ ¬ g)
    - AF f = A (true U f)
    - EF f = E (true U f)
    - AG f = ¬ E (true U ¬ f)
    - EG f = ¬ A (true U ¬ f)

Computation Tree Logic

- La verità di una formula è definita rispetto ad una struttura di Kripke (S,R,L) dove R è l'insieme delle coppie di stati (s,t) tali che t è un successore immediato di s
- Un cammino per una struttura di Kripke (S,R,L) è una sequenza infinita di stati (s0, s1, …) tale che ogni successiva coppia di stati (si, si+1) è un elemento di R

Computation Tree Logic



SMV (Symbolic Model Verifier)

- E' un model checker (McMillan 1991)
- Utilizza OBDD → problema esplosione degli stati ridotto rispetto alla tecnica di Clarke ed Emerson
- Permette di descrivere il modello con un linguaggio proprio
- Permette di definire proprietà in LTL e CTL
- Contro-esempio se la proprietà è falsa

SMV

- Limitazioni
  - Esplosione degli stati ridotta ma non risolta
  - Non sempre fornisce risposta
- Alternativa a BDD
  - Bounded Model Checking
    - Utilizza sat solver per confutare funzioni booleane
    - Fornisce contro-esempi di lunghezza limitata
    - Se una proprietà è vera non lo sapremo mai!

SMV

```
#include "nome_file_descrizione.smv"
module main ()
{
   VAR
   param_1, …, param_k: tipo;
   param_k+1, …, param_n: tipo;
   istanziazione del dispositivo da testare
   definizione delle proprietà
}

Le proprietà devono essere specificate come formule
logiche temporali
Ad esempio in LTL si usa la seguente sintassi
nome_proprietà: assert P;
```

Esempio

Bus arbiter

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1;
  ack2 := req2 & ~req1; -- favorisce processo 1 (req 1)


  Proprietà da verificare
  -----------------------

  mutex  : non posso avere entrambi ack veri
  serve  : se uno dei due req è vero allora almeno uno dei due ack deve
           essere vero
  waste1 : ack1 deve essere vero solo a fronte del fatto che req1 è vero
  waste2 : ack2 deve essere vero solo a fronte del fatto che req2 è vero
}
```

Esempio

- `mutex: assert ~(ack1 & ack2);`
- `serve: assert (req1 | req2) ->`
  `(ack1 | ack2);`
- `waste1: assert ack1 -> req1;`
- `waste2: assert ack2 -> req2;`

Proprietà valide solo a t=0

Esempio

- `mutex: assert G(~(ack1 & ack2));`
- `serve: assert G((req1 | req2) -> (ack1 | ack2));`
- `waste1: assert G(ack1 -> req1);`
- `waste2: assert G(ack2 -> req2);`

Proprietà valide sempre

Esempio

- Starvation del processo a priorità più bassa (req 2)?
  - `no_starve : assert G F (~req2 | ack2);`

no_starve è falsa!

Esempio

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;
  bit : boolean;

  next(bit) := ack1; -- valore di bit in t+1 = al valore di ack1 in t

  if (bit) {
    ack1 := req1 & ~req2;
    ack2 := req2;
  }
  else {
    ack1 := req1;
    ack2 := req2 & ~req1;
  }
```

no_starve è vera!

Riferimenti per SMV/NuSMV

- http://www-2.cs.cmu.edu/~modelcheck
- http://www-cad.eecs.berkeley.edu/~kenmcmil
- http://nusmv.irst.itc.it/

Linguaggi per la specifica di asserzioni

- VHDL
- Open Verification Library (OVL)
- SystemVerilog Assertion (SVA)
- OpenVera
- SystemC Verification Library (SVL)
- Property Specification Language (PSL)

VHDL

- Sviluppato nei primi anni 80 da IMB, Texas Instruments, Intermetrics, …
  - 1981: Requisiti iniziali parlano di supporto per gestione eccezioni ma non di asserzioni
  - 1983: introdotto il concetto di asserzione
    - sequenziale e concorrente
    - no operatori temporali
    - utili per esprimere proprietà invarianti tra variabili/ segnali
    - utili per documentazione
  - Recentemente è stato introdotto il PSL

Open Verification Library

- Libreria di checker scritti in VHDL, Verilog e SystemVerilog
  - di fatto asserzioni dichiarative verificate in simulazione
- Sviluppata per fornire
  - meccanismi di controllo delle asserzioni
  - reporting sulle asserzioni violate
  - meccanismi di reset
- I progettisti possono istanziare monitor OVL dentro a codice RTL

Open Verification Library

- Per specificare condizioni
  - che devono essere sempre soddisfatte
  - che non devono mai occorrere
  - sulla validità dei dati (es. range, pari, dispari, etc.)
  - sul cambio di un valore (es. incremento o decremento dentro un range)
  - sulla temporizzazione degli eventi (es. Entro un certo numero di cicli di clock
  - sulla validità di un protocollo
  - …

SystemVerilog Assertion

- Verilog non possiede il concetto di asserzione come il VHDL
- SVA sono state sviluppate per fornire ai progettisti un modo conciso e chiaro di esprimere proprietà sul design
- Si basa su un sottoinsieme della logica LTL (solo operatore G) a cui aggiunge le espressioni regolari
- Supporta asserzioni procedurali e dichiarative

OpenVera

- Nato come Vera nel 95, è un linguaggio imperativo e concorrente per creare testbench per simulazioni Verilog
  - Supporta generazione probabilistica di stimoli e monitoraggio di segnali/variabili a run time per misurare coverage
  - Successivamente esteso per supportare VHDL

OpenVera

- Comprato da Synopsys e reso pubblico sotto il nome di Open Vera nel 2001
  - Esteso con l'aggiunta del linguaggio ForSpec di Intel per supportare ABV
- Nel 2003 gran parte di OpenVera 1.3 è stato inserito in SystemVerilog 3.1

SystemC Verification Library

- Classe C++ sopra SystemC
- Introdotto nel dicembre del 2002
- Disponibile su Unix/Linux e Windows
- Permette
  - Introspezione dei tipi di dato SystemC
  - Generazione di stimoli con distribuzioni di probabilità parametrizzabili
  - Registrazione delle transazioni
- Non supporta asserzioni temporali

Property Specification Language

- Estensione di IBM Sugar per specificare proprietà temporali in modo rigoroso e con una semantica più formale rispetto a OpenVera
- E' basato su CTL e LTL, e permette di esprimere sia proprietà safety che liveness
  - Liveness verificabile solo formalmente, safety anche via simulazione

Property Specification Language

- E' diviso in quattro livelli (layers)
  - Boolean layer
    - Espressioni booleane sui segnali del design
  - Temporal layer
    - Per definire proprietà che valgono nel tempo
  - Verification layer
    - Per definire direttive per i tool di verifica (assert, assume, verification unit, …)
  - Modeling layer
    - Permette di inserire codice VHDL, Verilog, SystemVerilog o GDL in una specifica PSL per modellare l'ambiente in cui effettuare la verifica

Property Specification Language

- Molti tool di simulazione sono caratterizzati da *“single trace and monotonically advancing time”*
  - PSL simple subset
    - Sottoinsieme del PSL che può essere supportato facilmente sia in simulazione che formalmente