Applicazione di algoritmi evolutivi al collaudo ed alla verifica di circuiti digitali descritti ad alto livello

Gianluca Cumani

Indice

INDICE **2**

PREFAZIONE **4**

INTRODUZIONE **8**

1 FLUSSO DI PROGETTO BASATO SU VHDL **12**

1.1 VHDL 13
1.2 LA SINTESI 17
1.3 IL COLLAUDO 22

2 IL PROBLEMA DEL COLLAUDO **27**

3 RAGE **32**

3.1 ALGORITMI GENETICI 32
3.1.1 Il modello iniziale: l'evoluzione naturale *32*
3.1.2 Uno sguardo generico agli algoritmi genetici *34*
3.2 LA SIMULAZIONE 36
3.3 MODELLO DI GUASTO 38
3.4 L'ALGORITMO 40
3.4.1 BRAHMA: la creazione *41*
3.4.2 VISNU: l'evoluzione *42*
3.4.3 SHIVA: la selezione *46*
3.4.4 Casi particolari *46*
3.4.5 ARMAGEDDON: la reincarnazione *47*
3.5 IMPLEMENTAZIONE 48

4 APPLICAZIONI DI RAGE **56**

4.1 ANALISI DI TESTABILITÀ 56

4.1.1 Come usare RAGE per l'analisi di testabilità ... *56*
4.1.2 Risultati ottenuti usando RAGE per l'analisi di testabilità ... *58*
4.2 USARE RAGE ... 62
4.3 ATPG – AUTOMATIC TEST PATTERN GENERATOR ... 63
4.3.1 Usare RAGE come ATPG ... *63*
4.3.2 Risultati ottenuti usando RAGE come ATPG ... *64*
4.4 VERIFICA DI EQUIVALENZA ... 70
Come usare RAGE per la verifica di equivalenze ... *71*
4.4.1 Risultati ottenuti usando RAGE per la verifica di equivalenze ... *73*
CONCLUSIONI ... **76**
BIBLIOGRAFIA ... **78**
INDICE ANALITICO ... **80**
INDICE DELLE FIGURE ... **82**
INDICE DELLE TABELLE ... **83**

Prefazione

Il concetto di collaudo dei circuiti digitali è stato tradizionalmente riferito e utilizzato in descrizioni a basso livello. Per contro, grazie all'evoluzione degli strumenti di sintesi automatica, molte altre attività di progetto sono state trasferite da livello gate a livello RT; queste attività comprendono l'implementazione dell'algoritmo, la sintesi e la valutazione delle performance del circuito così realizzato.

La principale ragione per cui il collaudo è rimasto ancorato a descrizioni a basso livello è dovuta principalmente alla carenza di modelli di guasto significativi applicabili a descrizioni ad alto livello; questo ha inoltre portato alla mancanza di algoritmi di generazione automatica dei test *pattern* e di strumenti atti ad applicarli.

La difficoltà risiede nel fatto che si vorrebbero dei modelli di guasto per generare dei test pattern ad alto livello e che questi *pattern* garantissero elevate coperture anche rispetto ai modelli a basso livello.

La possibilità di generare test *pattern* ad alto livello produce numerosi vantaggi:

- la possibilità di conoscere nelle prime fasi di progetto quale sarà la copertura dei guasti del circuito permette di apportare eventuali modifiche al circuito, per migliorarne la collaudabilità, in tempi brevi con benefico impatto sul *time to market*

- viene offerta al progettista una migliore conoscenza delle parti non testabili; queste, infatti, sono evidenziate in un linguaggio comprensibile e non in un insieme di porte interne di difficile interpretazione
- la generazione dei vettori di collaudo può trarre vantaggio dalla scomposizione del progetto in processi separati e quindi generare dei test *pattern* più efficienti.

Alcuni circuiti, inoltre, contengono parti la cui descrizione a livello *gate* non è accessibile ma il cui funzionamento comportamentale è fornito oppure è facilmente ricostruibile; in questi casi il collaudo ad alto livello è l'unico possibile.

Questa tesi presenta un nuovo approccio al collaudo ad alto livello grazie al quale i test *pattern* generati offrono una buona testabilità anche quando applicati al *layout* sintetizzato. Viene inoltre proposta una misura di testabilità che si ritiene sia strettamente correlata alla *fault coverage* a basso livello.

Quando si ha a che fare con la generazione di vettori di collaudo per descrizioni ad alto livello, il primo problema da affrontare consiste nel definire dei modelli di guasto che abbiano delle proprietà ben definite. Occorre creare una buona correlazione tra la *fault coverage* ottenuta con il modello ad alto livello e quella ottenibile con misure effettuate con metodi classici. Questo modello di guasto deve tenere conto sia dei dati sia dei controlli, ovvero delle istruzioni componenti il codice. Occorre inoltre che i guasti così descritti risultino semplici da simulare, così da ridurre il tempo di calcolo necessario per eccitarli. Infine, il modello deve essere applicabile a progetti di elevata complessità ed di grandi dimensioni.

Il modello di guasto si basa sulla possibilità di eseguire le istruzioni. L’algoritmo adottato prevede l'esecuzione di tutte le istruzioni che compongono il codice VHDL. Ogni istruzione sarà composta da operandi e operatori che saranno sollecitati e quindi testati ogni qualvolta questa operazione venga eseguita. Quando un'istruzione è eseguita un numero prestabilito di volte, questa viene

considerata collaudata. Successivamente vengono eliminate (fase di *dropping*) anche le altre operazioni eseguite un numero sufficiente di volte.

L'algoritmo proposto si rivela utile alla soluzione di questi ed altri problemi. Si è rivelato utile infatti anche per la verifica di equivalenza tra due descrizioni in VHDL. L'implementazione dell'algoritmo è basata su un simulatore sviluppato dall'Università di Pittsburgh. Il codice del simulatore è stato successivamente da noi modificato ed integrato con algoritmi di generazione di sequenze di collaudo per renderlo atto allo scopo.

La scelta dell'algoritmo di generazione di *test vector* è ricaduta sugli algoritmi genetici. La possibilità di un collaudo esaustivo, infatti, è parsa subito inverosimile; richiederebbe infatti un tempo di calcolo assolutamente insensato. Un algoritmo del tipo *hill-climbing* è stato provato per breve tempo ma portava all'individuazione di massimi di copertura locali e non assoluti. La creazione dei *pattern* attraverso un algoritmo di tipo casuale forniva risultati accettabili in circuiti di piccole dimensioni ma risultava del tutto inefficiente in circuiti dalla notevole complessità e tortuosità descrittiva.

L'uso di algoritmi genetici ha permesso invece il raggiungimento di risultati di notevole qualità attraverso un uso razionale e modesto delle risorse di sistema. Il risultato ottenuto è un tool capace di funzionare su un elevato numero di descrizioni in VHDL e capace di fornire risultati che potessero confermare la bontà dell'algoritmo da cui trae origine. Questi risultati sono poi stati confrontati con quelli ottenuti da ATPG commerciali applicati alla descrizione a basso livello del medesimo circuito. Questo tool prende il nome di RAGE *(RT-level genetic Algorithm for test pattern GEneration*).

I risultati ottenuti utilizzando questo algoritmo e il prototipo su di esso basato sono risultati comparabili ed in alcuni casi migliori di quelli ottenibili mediante l'utilizzo di tecniche convenzionali e di prodotti commerciali. La sua efficacia ha permesso il suo sfruttamento in ambito europeo (Progetto Europeo Esprit FOST) come strumento di supporto per la verifica formale di equivalenza e di proprietà,

ed ha portato alla pubblicazione di un articolo presso *l'International Test Synthesis Workshop* nel maggio del 1997 e di uno presso l'*International Test Conference* nel Novembre 1997.

Questa tesi è stata svolta da Federico Cerati e Gianluca Cumani, studenti iscritti al corso di laurea in Ingegneria Elettronica del Politecnico di Torino, nell'ambito del gruppo CAD del dipartimento di Automatica ed Informatica.

Il lavoro di ricerca, codifica e sperimentazione necessario allo svolgimento di questa Tesi è stato equamente diviso fra i due laureandi.

In particolare Federico Cerati si è occupato della creazione della parte dell'algoritmo genetico adibita alla creazione e riproduzione degli individui e della codifica della parte adibita alla simulazione del prototipo.

Contemporaneamente Gianluca Cumani si è occupato della creazione di quella parte dell'algoritmo genetico che valuta e seleziona gli individui e della codifica del compilatore che rende simulabile una descrizione di un circuito.

Dopo la creazione del prototipo tutte le misure e gli esperimenti svolti per tarare l'algoritmo genetico in tutte le sue parti e per tutte le esigenze richieste, sono state svolte da entrambi.

Introduzione

Uno dei desideri principali dell'uomo è quello di migliorare la qualità della propria vita. Quest'impulso, unito al desiderio innato di conoscenza, ha spinto l'uomo alle conquiste più importanti della sua breve storia. Le conquiste della medicina hanno permesso di ottenere una vita più lunga e più sana, lo sfruttamento intensivo dell'agricoltura consente di non aver più problemi nella ricerca del cibo, lo sviluppo dei mezzi di locomozione consente l'esplorazione e la colonizzazione del mondo che ci circonda e la nascita dei sistemi di telecomunicazione ha permesso la globalizzazione delle informazioni. L'uomo ha inoltre sviluppato forme di svago articolate e complesse che possano rallegrarlo e divertirlo. Migliorare la qualità della vita significa non solo una riduzione della fatica o il mantenimento della propria salute fisica ma significa anche avere più tempo da dedicare a se stessi, alla propria realizzazione come esseri umani. I limiti dello spazio nel quale viviamo paiono essere lontanissimi e ci accingiamo ad ottenerne sempre di nuovi. Il tempo che ci è stato concesso invece è limitato e i suoi limiti sono, ahimè, visibili a tutti. Risparmiare tempo e spazio sono quindi obbiettivi di primaria importanza. Si cerca perciò di realizzare strumenti sempre più piccoli e sempre più veloci e di migliorarne le caratteristiche in modo da assicurare migliori prestazioni. Tutto questo porta ad una maggiore soddisfazione di chi usa i prodotti. Si è cioè innalzata la qualità dei prodotti stessi.

Un fattore che contribuisce in maniera essenziale alla qualità di un oggetto consiste nella capacità dell'oggetto di adempiere allo scopo per il quale è stato pensato e costruito. Se ciò non avviene, il prodotto è soggetto ad un difetto. Per

migliorare la qualità quindi è necessario rilevare i difetti ed i guasti che possono inficiare il normale utilizzo del manufatto ed eliminarli. L'insieme di accorgimenti e di tecniche che mirano ad evidenziare la possibile presenza di guasti prende il nome di collaudo. La necessità di collaudare è sempre esistita ma la sua realizzabilità si è fatta via via più complicata con l'aumentare della complessità dei prodotti. Oggetti come le dighe, i motori di automobili sportive, gli schermi al plasma, le apparecchiature per le TAC sono solo esempi di sistemi complessi e di difficile collaudo. Nell'ambito strettamente elettronico, la nascita dei processi di integrazione ha portato allo sviluppo di circuiti digitali sempre più complessi e miniaturizzati; un microprocessore moderno contiene su una superficie di pochi millimetri quadrati diversi milioni di transistor. Collaudare questi transistor è un'impresa ardua. Fino ad oggi le tecniche che permettevano questo collaudo si basavano sulla conoscenza approfondita della struttura del circuito in esame. Ciò comportava un notevole utilizzo di risorse umane e di tempo per ottenere un modello del circuito strettamente correlato alla sua realtà realizzativa sul quale effettuare le prove di collaudo. Per decidere quali prove effettuare, ovvero quale sequenza di ingressi applicare per verificare il corretto funzionamento del circuito, era necessario descriverlo componente per componente o addirittura transistor per transistor; tutto ciò avveniva quindi molto dopo il periodo di progetto e negli istanti immediatamente precedenti alla realizzazione fisica e commerciale del circuito. Qualora i risultati non fossero poi stati all'altezza di quelli preventivati, si sarebbe dovuto procedere ad una modifica del progetto e ad una ripetizione di tutti gli stadi di sviluppo fino alla nuova realizzazione circuitale.

L'idea che viene sviluppata in questa tesi, prevede la possibilità di avvicinare il momento della verifica della collaudabilità a quello della progettazione, riducendo così i tempi di un eventuale riprogettazione e riducendo il costo della stessa mantenendo però la validità e la credibilità dei risultati ottenibili. Per far ciò si è scelto di operare non su descrizioni a basso livello del circuito, descrizioni cioè basate sulla topologia delle porte logiche o dei transistor, ma su una descrizione di

tipo comportamentale del circuito in esame, una descrizione cioè ad alto livello. Questo rende la verifica della collaudabilità una parte integrante del processo di progetto e non del processo di realizzazione come ora avviene. Tutto ciò permette di ridurre notevolmente i tempi di riprogettazione e quindi riduce i costi del collaudo. Il progettista avrà inoltre informazioni dirette e facilmente leggibili riguardanti la collaudabilità di ciò che in quel momento produce; infatti i risultati sono espressi in un linguaggio ad alto livello, facilmente comprensibile e non sotto forma di insiemi di porte logiche sparse lungo tutto il circuito. Tutto ciò significa aumentare la qualità della vita del progettista. Poiché i risultati dovranno essere validi quanto quelli ottenibili per vie tradizionali, la qualità del prodotto finale sarà incrementata grazie al risparmio di tempo e di denaro ottenuto nella sua realizzazione. Si avranno quindi oggetti meno cari o utili più elevati e questo aumenterà anche la qualità della vita dei compratori o degli investitori.

In questo contesto RAGE si propone come uno strumento che, permettendo sia l'analisi della testabilità delle descrizioni VHDL sia la creazione di sequenze di collaudo, semplifichi e velocizzi la fase di progetto. Propone un modello di guasto che si basa sulla possibilità di eseguire le istruzioni, prevede quindi l'esecuzione di tutte le istruzioni che compongono il codice VHDL. Al fine di individuare i vettori di collaudo utilizza le potenzialità offerte dagli algoritmi genetici.

In questa tesi si tratteranno gli elementi essenziali della progettazione di circuiti digitali; si offrirà quindi una descrizione del linguaggio utilizzato per lo sviluppo dei circuiti, ovvero il VHDL, e dei suoi pregi. Saranno poi citate le principali metodologie grazie alle quali è possibile ottenere il layout di un circuito partendo da una sua descrizione ad alto livello; sarà cioè introdotto il concetto di sintesi automatica. Successivamente si dimostrerà la necessità del collaudo dei circuiti e ci si accosterà alle problematiche relative ad esso. Sarà quindi proposto un algoritmo che permette un nuovo tipo di approccio al problema del collaudo; poiché parte di questo algoritmo prevede l'uso di algoritmi genetici, verrà fornita una panoramica su di essi. L'algoritmo sarà quindi spiegato in maniera dettagliata e ne sarà fornita una possibile implementazione. Infine verranno presentati i

risultati ottenuti applicando il software prodotto su un numero significativo di esempi.

1 Flusso di progetto basato su VHDL

Progettare un circuito significa dare una descrizione dello stesso che sia comprensibile e che ne rispecchi il comportamento. Questa descrizione può essere più o meno dettagliata a seconda delle esigenze del momento del progettista. Per la produzione di prototipi da far visionare ad un committente è infatti utile creare delle descrizioni sommarie che ne simulino il comportamento funzionale in modo da fornire un'idea di ciò che sarà il prodotto finale.

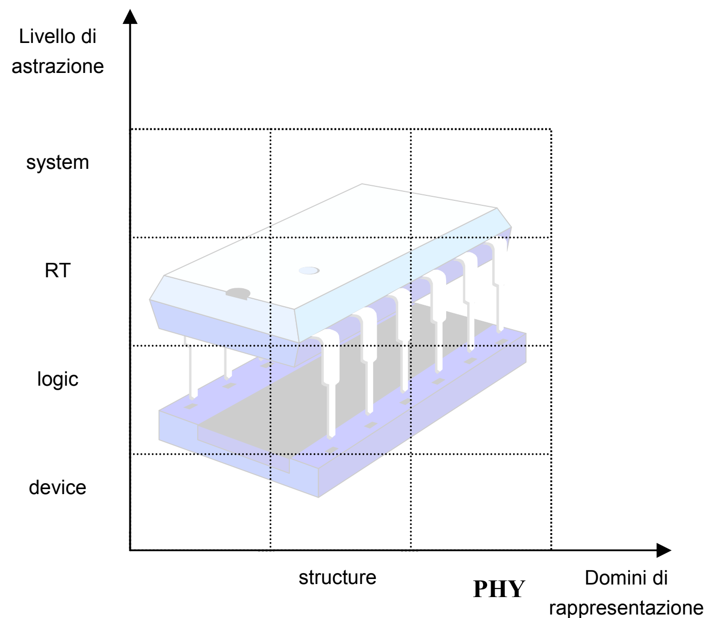

Figura 1: ***Livelli di astrazione e domini di rappresentazione.***

Qualora si vogliano creare modelli per esigenze diverse, si dovrà porre maggiore attenzione ad aspetti differenti del circuito. Si dovrà, ad esempio,

mettere in risalto come i componenti del circuito si interconnettono tra loro in modo da poter evitare la creazioni di ponti al momento della produzione. Sono quindi possibili diverse metodologie di progetto, rappresentabili in una matrice che prende il nome di dominio di rappresentazione. Il dominio di rappresentazione è quindi l'insieme di particolari aspetti considerati interessanti in fase di progetto e sviluppo. Sfruttando questi domini si può progettare descrivendo il comportamento del sistema, behavior, la sua topologia in termini di interconnessione tra blocchi funzionali considerati elementari che eseguono un sottoinsieme delle operazioni definite nel dominio precedente, structure, e la sua costituzione fisica in termini di componenti elementari che implementano i blocchi definiti nel dominio *structural*, physical.

Il livello di astrazione rappresenta invece il grado di dettaglio che si desidera, dai processi e algoritmi, system, alle operazioni logiche e aritmetiche, RT, fino alle equazioni logiche, logic, oppure alle tensioni e correnti, device. Un buon supporto alla progettazione deve fornire la possibilità di progettare posizionandosi in una qualunque di queste celle a seconda della necessità.

1.1 VHDL

Il VHDL (*Very high speed integrated circuits Hardware Description Language*) è un linguaggio utilizzato per descrivere dei generici circuiti digitali indipendentemente dalla tecnologia utilizzata e dalle metodologie di progetto. Infatti la base del linguaggio è una astrazione della natura dell'hardware digitale, astrazione che include però il comportamento, le tempistiche e le caratteristiche strutturali dei dispositivi descritti.

Il VHDL è stato sviluppato pensando ai molteplici problemi che si presentano nel progetto, nello scambio di informazioni e nella necessità di documentare l'hardware digitale. Per esempio, una tipica consegna di hardware al Governo degli Stati Uniti contempla la fornitura di migliaia di pagine di documentazione a partire dalla commessa fino al collaudo del componente. Quando un componente

deve poi essere sostituito è necessario un grande sforzo per ricostruirne il presunto comportamento. Un buon linguaggio di descrizione hardware risolve questi problemi poiché la documentazione è eseguibile e tutti gli elementi costitutivi sono uniti in un unico modello.

Nonostante la presenza di numerosi linguaggi di descrizione dell'hardware, nessuno di essi era mai stato accettato come standard dall'industria prima dell'avvento del VHDL. I linguaggi precedenti, infatti, erano strettamente legati alla tecnologia usata o erano legati a particolari simulatori ed alle loro case produttrici. Il VHDL permette invece di scegliere liberamente la tecnologia e le metodologie da utilizzare utilizzando sempre il medesimo linguaggio; inoltre, il livello di astrazione del VHDL è tale da facilitare l'inserimento di nuovi dispositivi e future tecnologie nei progetti attuali.

Il VHDL offre diversi vantaggi rispetto agli altri linguaggi.

- *Pubblica disponibilità:* il VHDL è stato sviluppato sotto specifiche del governo degli Stati Uniti ed ora è uno standard IEEE. Il governo statunitense ha quindi un forte interesse a mantenere il VHDL come standard universale.
- *Metodologie di progetto e supporto di varie tecnologie:* il VHDL può supportare differenti metodologie di progetto (es. *top - down* oppure *bottom - up*) e differenti tecnologie (es. circuiti sincroni o asincroni, PLA[1] oppure logica sparsa). In tal modo il VHDL può essere utilizzato da industrie che operano in ambiti differenti e con problemi progettuali diversi. Il VHDL è adatto alle *software house* che vendono librerie di CAD/CAE[2] così come alle industrie aerospaziali che realizzano un considerevole numero di ASIC[3].

---

[1] PLA: Programmable Logical Array
[2] CAD/CAE: Computer Aided Design/Computer Aided Engineering
[3] ASIC: Application Speed Integrated Circuit

- *Indipendenza dalla tecnologia e dai processi di realizzazione:* il VHDL non è legato ad alcuna tecnologia particolare né ad un determinato processo realizzativo; una descrizione simulabile del sistema può essere sviluppata ad un livello astratto e successivamente sintetizzata a livello *gate* a seconda del processo realizzativo scelto (CMOS, nMOS, GaAs). Elevata capacità descrittiva: il VHDL permette rappresentazioni circuitali a differenti livelli di complessità; riconosce quindi descrizioni puramente comportamentali così come descrizioni a livello *gate*. Uno dei principali vantaggi del VHDL consiste nel rappresentare una operazione di un sistema digitale e simularne il comportamento ad uno qualunque di questi livelli. È inoltre possibile simulare un dispositivo che presenti in alcuni sottosistemi delle descrizioni ad alto livello ed in altri sottosistemi presenti invece una descrizione più accurata. Tutto ciò permette lo sviluppo di un progetto che rifletta correttamente le specifiche iniziali e che sia altresì facilmente migliorabile consentendo di intervenire direttamente sul singolo blocco da modificare.

- *Scambio di progetti:* poiché il VHDL è uno standard, tutti i circuiti descritti in questo linguaggio possono essere simulati su un qualunque simulatore VHDL conforme allo standard medesimo. Ciò significa che un modello sviluppato da una azienda funzionerà anche presso altre aziende che utilizzano differenti programmi di simulazione e quindi può permettere la creazione di librerie di componenti che ogni progettista potrà utilizzare liberamente all'interno di un qualunque simulatore. Questo inoltre facilita lo scambio di informazioni presso i differenti gruppi di lavoro all'interno dell'industria permettendo uno sviluppo separato dei sottosistemi che realizzano il circuito. I sottosistemi possono essere eventualmente acquistati anche presso terzi maggiormente specializzati. Tutto ciò riduce gli sforzi necessari ad integrare le varie parti di un progetto.

- *Progetti su larga scala e riutilizzo dei progetti:* il VHDL è stato realizzato seguendo una filosofia comune ai moderni linguaggi di programmazione.

La possibilità di decomporre in sottosistemi i progetti che coinvolgano un elevato numero di sviluppatori è quindi importante quanto la capacità di descrizioni dettagliate ed accurate. La possibilità di creare packages, entità e processi è insita nel linguaggio facilitando così la possibilità di suddivisione del lavoro e la possibilità di sperimentare nuove soluzioni all'interno di un progetto già sviluppato. Questi elementi del linguaggio permettono inoltre un facile riutilizzo di progetti o parti di essi.

- *Richieste esplicite:* il governo statunitense richiede esplicitamente le descrizioni VHDL di ogni ASIC acquistato presso industrie pubbliche o private. Sulla scia del Governo americano si stanno muovendo anche molte industrie private che, avendo acquisito familiarità con il VHDL producendo circuiti per il governo, richiedono a loro volta le descrizioni dei circuiti comperati ai loro fornitori.

Tutte queste motivazioni fanno sì che la scelta del VHDL come linguaggio per la descrizione dei circuiti hardware sia una scelta ormai obbligata per garantire l'universalità e la facile diffusione e comprensione del proprio lavoro.

1.2 La sintesi

La sintesi automatica è la trasformazione, attraverso opportune regole, di una descrizione di un sistema hardware, intesa come specifica, in un'altra descrizione dello stesso sistema, intesa come implementazione in termini di blocchi considerati elementari.

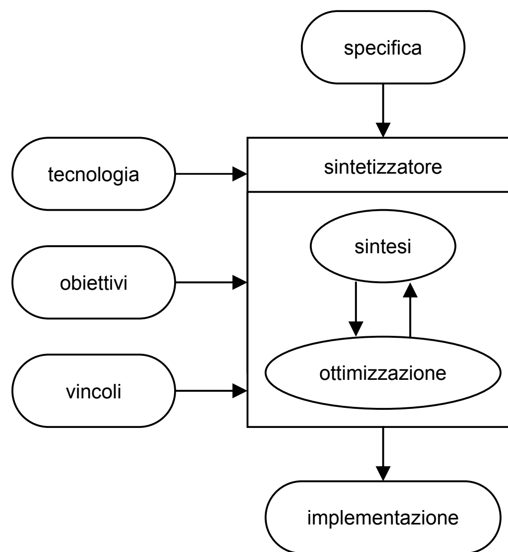

Figura 2: ***Processo di sintesi.***

Questo processo dipende dalla tecnologia nella quale si vuole realizzare l'implementazione, dagli obiettivi che si pone il progettista e dai vincoli imposti dall'ambiente.

Con tecnologia viene indicato il tipo di realizzazione fisica specificandone le peculiarità e fornendo i blocchi elementari utilizzabili e i loro parametri.

Gli obiettivi specificano l'importanza dei diversi parametri di costo con cui valutare la bontà di un'implementazione, fra questi la velocità, l'area occupata dal circuito, il consumo e la collaudabilità.

I vincoli specificano le condizioni imposte da tutto ciò che circonda il circuito, come il tempo di arrivo dei valori in ingresso o il massimo ritardo di quelli in uscita.

A sintesi ultimata si vuole ottenere un implementazione che sia coerente con le specifiche, realizzabile nella tecnologia specificata, di costo minimo e che rispetti i vincoli.

Per il progettista la coerenza con le specifiche significa che ciò che viene sintetizzato deve essere ciò che lui ha specificato, mentre gli strumenti di sintesi assumono che il comportamento desiderato sia quello ottenibile dalla simulazione.

Questo significa che le specifiche devono essere formalizzate in un linguaggio simulabile. La semantica dei linguaggi di descrizione, di contro, è spesso ambigua; alcune descrizioni, se simulate, esibiscono comportamenti dissimili da quelli dell'hardware sintetizzato (es. una porta logica in fase di simulazione non presenta ritardo tra l'ingresso e l'uscita mentre una volta sintetizzata questo è presente).

Occorre quindi definire un sottoinsieme del linguaggio che sia sintetizzabile, degli stili di descrizione in grado di eliminare le ambiguità e un interpretazione per la sintesi di alcune istruzioni del linguaggio (es. le assegnazioni a ritardo nullo).

Il rispetto dei vincoli richiede che il sintetizzatore sia in grado di valutare accuratamente i ritardi di propagazione, identificare i cammini critici e ottimizzare il ritardo di quest'ultimi. Tale ottimizzazione cerca di raggiungere un compromesso tra l'area complessiva occupata dal circuito sintetizzato e il ritardo di propagazione sui cammini.

Il sintetizzatore, per la realizzabilità tecnologica, deve essere in grado di realizzare qualsiasi funzione logica con un insieme di primitive variabile di tecnologia in tecnologia.

Trovate tutte le implementazioni che soddisfano i parametri succitati, al simulatore è richiesto di scegliere quella ottima, ovverosia quella che minimizza una funzione di costo globale.

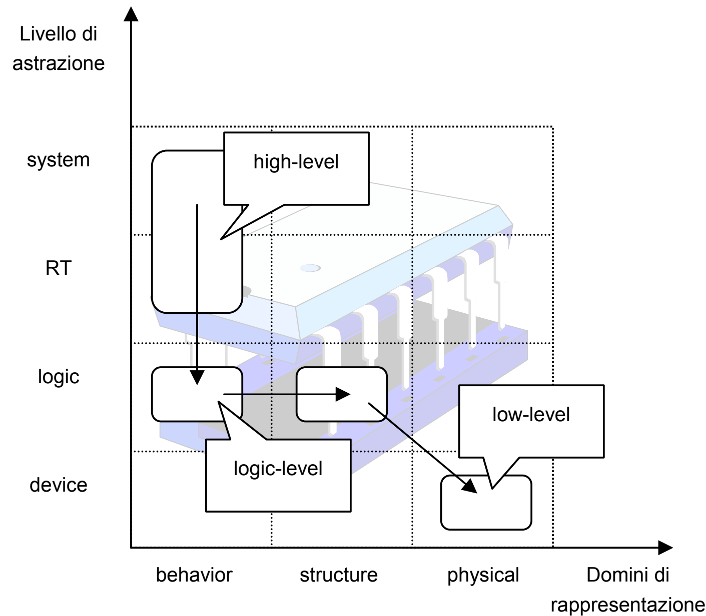

Figura 3: ***Classificazione delle possibili sintesi.***

Si possono identificare diversi tipi di sintesi in base ai domini di rappresentazione e ai livelli di astrazione interessati, otteniamo così tre livelli di sintesi:

- *High-level*: fornisce una descrizione a livello logico, partendo da descrizioni comportamentali a più alto livello.
- *Logic-level*: fornisce una descrizione strutturale al livello logico, partendo da una descrizione comportamentale allo stesso livello.
- *Low-level* o *Silicon compilation*: fornisce una descrizione del *layout*, partendo da una descrizione a livello logico.

I diversi strumenti di sintesi, classificati in base al livello di astrazione a cui è fornita la specifica, sono caratterizzati dal tipo di descrizione accettata come specifica e da quella generata come implementazione nonché dagli specifichi problemi di ottimizzazione e dei relativi algoritmi risolutivi.

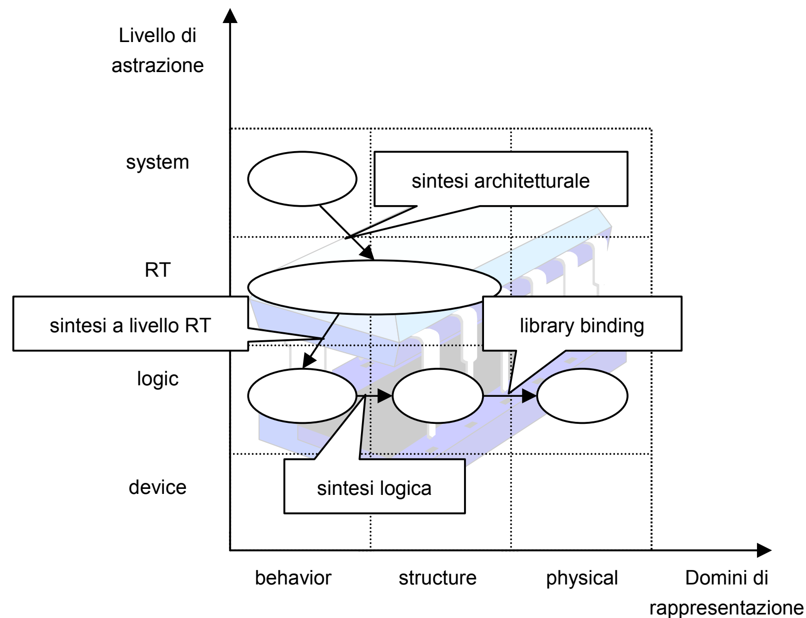

Figura 4: ***Classificazione degli strumenti di sintesi.***

Si possono avere così strumenti per la sintesi architetturale, che da un algoritmo o un *data-flow* creano delle unità funzionali interconnesse ottimizzate mediante *scheduling, allocation* e *resource sharing* usando algoritmi di manipolazione del *control/data-flow graph*, la sintesi a livello RT, che implementa la *Control Unit* partendo da una FSM complessa a livello RT-behavioral, la sintesi logica, che da un insieme di equazioni logiche genera delle porte logiche interconnesse ottimizzando con manipolazione algebriche sia il loro numero che il *fanout*, e library binding, che converte un insieme di porte logiche in celle interconnesse ottimizzando l'area occupata e i ritardi con algoritmi *tree matching*.

Sul mercato sono disponibili alcuni strumenti di sintesi *high-level* a partire da dialetti VHDL, mentre quelli di *logic-level* e *low-level* sono commercialmente

disponibili, efficienti e largamente usati in ambito industriale. Sintetizzatori in grado di generare automaticamente il *layout*, a partire da descrizioni comportamentali a livello sistema, sono disponibili solo per applicazioni molto specifiche. Strumenti di sintesi applicabili a tutti i livelli, in tutti i domini, per qualsiasi stile di descrizione, per qualsiasi metodologia di progetto e per qualsiasi architettura sono ancora un sogno, lontano dall'avverarsi.

1.3 Il collaudo

Il problema del collaudo di ciò che l'uomo produce risale alle stesse origini dell'umanità. Qualunque prodotto reale, infatti, è stato pensato al fine di adempiere ad uno scopo ed è necessario verificare la funzionalità di ogni oggetto costruito.

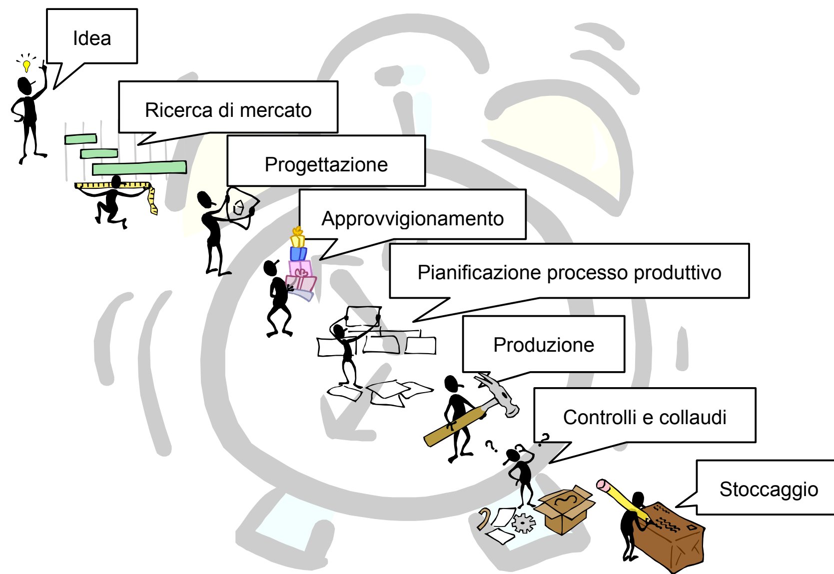

Figura 5: *Il time to market.*

Questa necessità è stata acuita dal processo industriale, processo che prevede la produzione di manufatti in grande scala e quindi in grande numero. Inoltre questi gli oggetti si sono via via evoluti diventando sempre più complessi e quindi difficili da collaudare.

Il collaudo è, infatti, l'insieme delle operazioni atte a valutare, con una certa probabilità, il corretto funzionamento di un sistema manufatto, secondo determinate specifiche, nel momento in cui viene effettuato il collaudo stesso. La cessazione dell'attitudine di un elemento ad adempiere alla funzione richiesta viene chiamato guasto e le circostanze che hanno portato al guasto ne sono le cause. Le cause di guasto possono essere di diversa natura; possono essere errori di progetto, di fabbricazione od anche il cattivo utilizzo. Il fine del collaudo è

quello di discriminare i pezzi mal funzionanti da quelli che invece adempiono correttamente alla loro funzione.

Il collaudo in fase di produzione permette un incremento della qualità del prodotto finale, una riduzione dei costi (assistenza, riparazione, sostituzione) ed inoltre riduce il *time to market*, ovvero il tempo necessario affinché un prodotto giunga sul mercato.

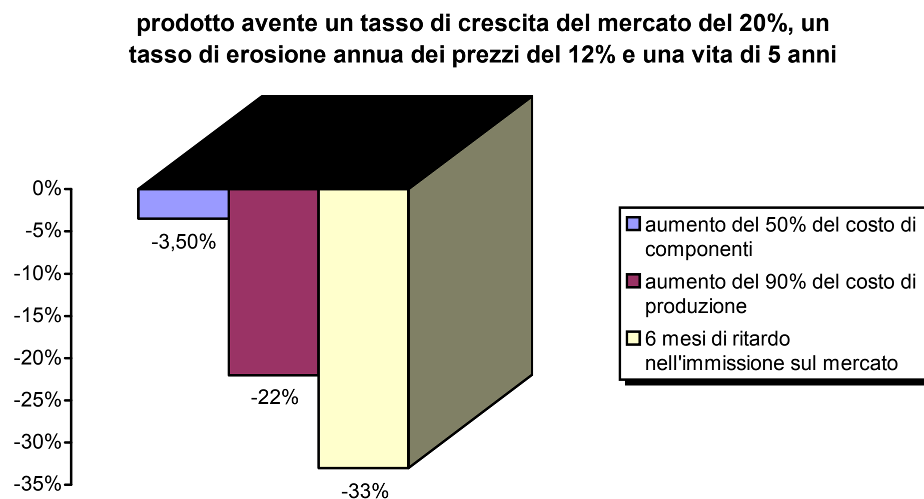

Figura 6: ***Impatto del time to market sui profitti.***

Il collaudo effettuato durante il ciclo di vita del prodotto permette invece di accorgersi presto della presenza di un malfunzionamento e quindi permetterne la riparazione prima dell'insorgere di danni più gravi.

Detto questo sembrerebbe sia conveniente collaudare ogni volta se ne presenti l'occasione e il più possibile; ma testare ha un costo molto elevato. L'aumento della densità dei transistor per millimetro quadro di silicio e l'aumento della frequenza di funzionamento rendono la testabilità sempre più costosa e difficoltosa. Inoltre il mercato richiede prodotti sempre più perfezionati e sempre più potenti in sempre minor tempo riducendo così il tempo di commerciabilità di un oggetto. Ciò che una volta rimaneva sul mercato per parecchi anni, ora viene

sostituito dopo pochi anni o addirittura dopo pochi mesi! È quindi necessario comprimere al massimo il *time to market* a discapito del tempo dedicato al collaudo.

Di conseguenza la possibilità di scoprire malfunzionamenti nelle prime fasi di progetto permettere di evitare laboriose riprogettazioni e modifiche. Scoprire un difetto nelle prime fasi di progetto ha un costo di riparazione di tre o più ordini di grandezza minore rispetto allo scoprirlo al momento della commercializzazione. Il compratore è inoltre sempre più esigente, richiede una qualità sempre superiore e quindi è sempre meno disposto a riparazioni o ad interventi sul campo.

Attualmente esistono quattro tipi di collaudo: il collaudo funzionale, il collaudo strutturale, il collaudo esaustivo ed il collaudo pseudo casuale.

Il collaudo esaustivo prevede l'applicazione, come sequenze (pattern) test, di tutte le possibili combinazioni degli ingressi, in ogni possibile stato del sistema. Questo approccio funziona solamente nel caso di circuiti semplici e con pochi ingressi altrimenti il testing diventa intollerabilmente lungo. Per un sommatore a 32 bit, ad esempio, il tempo necessario per effettuare un testing esaustivo, applicando un pattern ogni nanosecondo, ammonterebbe più di mille anni.

Il collaudo pseudo casuale prevede la generazione dei pattern in maniera casuale o pseudo casuale. Questa tecnica è la più usata in quanto è facilmente applicabile. Il collaudo non sarà certamente esaustivo ma si arresterà quando si sarà ottenuta una copertura ritenuta sufficiente. È quindi importante trovare i pattern che massimizzano la copertura e che siano i più corti possibile, per ridurre il tempo di test.

Il collaudo strutturale prevede che queste sequenze eccitino tutta una serie di guasti contenuti in una lista dei guasti (*fault list*) stabilita a priori. In questo modo si ha una elevata conoscenza della copertura e una elevata capacità diagnostica ma bisogna avere una elevata conoscenza della struttura fisica del circuito e quindi la lista dei guasti è correlata e dipendente dalla realizzazione del progetto.

Il collaudo funzionale, invece, tende a verificare il corretto funzionamento del sistema rispetto alle sue certifiche funzionali piuttosto che a certificare l'assenza di guasti. Un test di questo tipo non richiede la conoscenza tipologica dell'unità da testare, può essere effettuato alla velocità di clock di effettivo utilizzo ed è indipendente dai motivi di guasto. Fino ad ora però era difficile valutare l'effettiva bontà dei risultati ottenuti ed era quasi impossibile generare automaticamente i pattern da applicare. Infatti è impossibile trovare dei modelli di guasto validi a livello funzionale.

I modelli di guasto finora sviluppati sono sempre applicabili alla realtà fisica dell'oggetto e sono sostanzialmente di tre tipi: errori di tipo stuck at, errori di tipo bridge ed errori di ritardo.

Gli errori più comuni sono quelli di tipo stuck at; si ipotizza che un nodo della rete sia fisso ad un determinato livello logico. Il modello di errore di tipo bridge ipotizza invece la presenza di cortocircuiti non voluti tra coppie di nodi della rete. Infine il modello di ritardo tende a rappresentare gli errori che si verificano alla frequenza operativa di clock.

Il concetto di testabilità per i circuiti digitali è stato tradizionalmente definito e sfruttato solo a basso livello. D'altro canto, grazie all'evoluzione degli strumenti di sintesi, molte attività della progettazione si sono spostate dal livello *gate* a quello RT: la validazione, la sintesi, la valutazione delle prestazioni ecc.

La ragione principale per cui la testabilità non ha ancora intrapreso la stessa strada è da ricercarsi nella mancanza di modelli di guasti ad alto livello credibili e di algoritmi per la generazione di sequenze di test di alta qualità. L'ottimo sarebbe poter generare delle sequenze di test a livello RT che abbiano una buona copertura degli errori dati da un modello di guasti a basso livello.

I vantaggi della generazione di sequenze di test ad un livello più alto di quello *gate* si possono sintetizzare in:

- Minor tempo per la generazione delle sequenze, in quanto l'ATPG ha a che fare con una rappresentazione più astratta del progetto.

- Un interazione migliore con il progettista, in quanto i guasti non testabili si riferiscono a istruzioni o processi contenuti nella descrizione VHDL, quindi maggiormente comprensibili rispetto ad un insieme di porte logiche interne.
- Generazione di sequenze di ottima qualità sfruttando la divisione in processi della descrizione VHDL.
- Con una oculata scelta del modello di guasti nonché degli algoritmi che costituiscono l'ATPG si ottiene una buona copertura.

I vantaggi sopracitati sono ancora più palesi nei circuiti di grandi dimensioni, laddove la differenza di complessità tra una descrizione di alto livello e una di basso è più evidente.

2 Il problema del collaudo

Il concetto di collaudo dei circuiti digitali è stato tradizionalmente riferito e utilizzato in descrizioni a basso livello. Per contro, grazie all'evoluzione degli strumenti di sintesi automatica, molte altre attività di progetto sono state trasferite da livello *gate* a livello RT; queste attività comprendono l'implementazione dell'algoritmo, la sintesi e la valutazione delle performance del circuito così realizzato.

La principale ragione per cui il collaudo è rimasto ancorato a descrizioni a basso livello è dovuta principalmente alla carenza di modelli di guasto significativi applicabili a descrizioni ad alto livello; questo ha inoltre portato alla mancanza di algoritmi di generazione automatica dei test pattern e di strumenti atti ad applicarli.

La difficoltà risiede nel fatto che si vorrebbero dei modelli di guasto per generare dei test pattern ad alto livello e che questi pattern garantissero elevate coperture anche rispetto ai modelli a basso livello.

La possibilità di generare sequenze di collaudo partendo da descrizioni ad alto livello è stata recentemente campo di numerose ricerche. Sono stati proposti differenti approcci, ognuno di essi basato su differenti modelli comportamentali del sistema quali macchine a stati finiti, descrizioni gerarchiche, grafi di flusso di dati o semplice codice in un linguaggio di descrizione. Ogni approccio crea un modello di guasto valido per il modello comportamentale in esame.

La possibilità di generare test pattern ad alto livello produce numerosi vantaggi:

1. la possibilità di conoscere nelle prime fasi di progetto quale sarà la copertura dei guasti del circuito permette una eventuale riprogettazione in tempi brevi con benefico impatto sul time to market
2. viene offerta al progettista una migliore conoscenza delle parti non testabili; queste, infatti, sono evidenziate in un linguaggio comprensibile e non in un insieme di porte interne di difficile interpretazione
3. la generazione dei vettori di collaudo può trarre vantaggio dalla scomposizione del progetto in processi separati e quindi generare dei test pattern più efficienti.

Questi vantaggi dovrebbero risultare particolarmente evidenti in circuiti di grosse dimensioni dove i tradizionali ATPG mostrano i maggiori limiti e dove la differenza di complessità tra il circuito a basso livello e la sua descrizione ad alto livello è rilevante. Alcuni circuiti, inoltre, contengono parti la cui descrizione a livello *gate* non è accessibile ma il cui funzionamento comportamentale è fornito oppure è facilmente ricostruibile; in questi casi il collaudo ad alto livello è l'unico possibile.

Questa tesi presenta un nuovo approccio al collaudo ad alto livello grazie al quale ci si aspetta che i test pattern generati offrano una buona testabilità anche quando applicati al layout sintetizzato. Viene inoltre proposta una misura di testabilità che si ritiene sia strettamente correlata alla *fault coverage* a basso livello.

L'algoritmo proposto si rivela utile alla soluzione di questi ed altri problemi. Si è rivelato utile infatti anche per la verifica di equivalenza tra due descrizioni in VHDL. Poiché un linguaggio ad alto livello permette una certa libertà di espressione, uno stesso comportamento funzionale può essere descritto in modi differenti. Questa differenza si ripercuoterà anche sul layout sintetizzato creando due circuiti differenti da un punto di vista fisico. Questi circuiti così ottenuti

possono presentare differenze di performance anche rilevanti e quindi rendere preferibile una descrizione piuttosto che un'altra. Bisogna però verificare che le due descrizioni siano effettivamente equivalenti dal punto di vista funzionale. L'algoritmo proposto permette di rispondere anche a questo quesito con una notevole affidabilità.

Da ciò detto l'algoritmo deve operare su una descrizione in VHDL a livello RT eseguendo le istruzioni contenute nel codice, procedendo così alla sua simulazione.

La sua implementazione presenta due possibilità: la modifica della descrizione originale in maniera tale da rendere possibile l'acquisizione della informazioni atte al corretto funzionamento dell'algoritmo oppure la modifica del simulatore.

La prima strada consiste nell'inserire delle istruzioni aggiuntive all'interno del codice VHDL; l'esecuzione di queste istruzioni attesterebbe l'avvenuta attivazione del codice ad esse legato, ovvero di una porzione di descrizione che deve essere collaudata. Questa via permette l'uso dei simulatori commerciali ma aumenta il tempo necessario al raggiungimento dei risultati. Inoltre questa strada prevede la creazione di un tool separato che generi il codice modificato partendo da quello originario.

La seconda soluzione è certamente più pulita e veloce e prevede che sia il simulatore stesso ad acquisire le informazioni utili durante la simulazione del circuito. Ciò permette di ottenere un tool integrato capace di risolvere autonomamente il problema. La soluzione in questione inoltre richiede minor tempo di elaborazione ed è certamente preferibile per ottenere prodotti commerciali. Questa strada presenta due difficoltà: la possibilità di ottenere i sorgenti di un simulatore VHDL e la difficoltà tutt'altro che trascurabile di riuscire a comprendere e modificare con successo del codice complesso ed altamente ottimizzato. È stato acquistato dall'università di Pittsburgh il codice di un simulatore funzionante su un sottoinsieme significativo di istruzioni VHDL. Per avere un prototipo funzionante che implemento l'algoritmo descritto in questa

tesi, il codice è modificato ed integrato con algoritmi di generazione di sequenze di collaudo per renderlo atto allo scopo.

La scelta dell'algoritmo di generazione di sequenze di test è ricaduta sugli algoritmi genetici. La possibilità di un collaudo esaustivo, infatti, è parsa subito inverosimile; richiederebbe infatti un tempo di calcolo assolutamente insensato. Un algoritmo del tipo *hill-climbing* è stato provato per breve tempo ma portava all'individuazione di massimi di copertura locali non sufficienti.

La creazione dei pattern attraverso un algoritmo di tipo casuale fornisce risultati accettabili in circuiti di piccole dimensioni ma risultava del tutto inefficiente in circuiti della notevole complessità e tortuosità descrittiva.

L'uso di algoritmi genetici permette invece il raggiungimento di risultati di notevole qualità attraverso un uso razionale e modesto delle risorse di sistema. Il risultato ottenuto è un tool capace di funzionare su un elevato numero di descrizioni in VHDL e capace di fornire risultati che potessero confermare la bontà dell'algoritmo da cui trae origine. Questi risultati sono poi stati confrontati con quelli ottenuti da ATPG commerciali applicati alla descrizione a basso livello del medesimo circuito. Questo tool prende il nome di RAGE (*RT-level genetic Algorithm for test pattern GEneration*).

Quando si ha a che fare con la generazione di vettori di collaudo per descrizioni ad alto livello, il primo problema da affrontare consiste nel definire dei modelli di guasto che abbiano delle proprietà ben definite:

- deve esistere una buona correlazione tra la fault coverage ottenuta con il modello ad alto livello e quella ottenibile con misure effettuate con metodi classici
- il modello di guasto deve tenere conto sia dei dati sia del controllo
- i guasti così descritti devono risultare semplici da simulare
- il modello deve essere applicabile a progetti di elevata complessità ed di grandi dimensioni

Il modello di guasto adottato prevede l'esecuzione di tutte le istruzioni che compongono il codice in VHDL. Ogni istruzione sarà composta da operandi e operatori che saranno sollecitati e quindi testati ogni qualvolta questa operazione venga eseguita. Quando un'istruzione è eseguita un numero prestabilito di volte, questa viene considerata collaudata.

3 RAGE

RAGE (RT-level genetic Algorithm for test pattern GEneration) rappresenta un nuovo approccio per risolvere il problema della generazione di sequenze a livello RT tali da produrre una buona copertura qualora vengano applicate al circuito sintetizzato a livello *gate*. Si tratta di un algoritmo capace di generare sequenze, un vettore per ogni ciclo di clock, partendo da un sottoinsieme di VHDL sintetizzabile (si assume che la descrizione sia sincrona) mediante un algoritmo genetico.

3.1 Algoritmi Genetici

3.1.1 Il modello iniziale: l'evoluzione naturale

Gli algoritmi genetici sono stati inventati imitando i processi osservati nell'evoluzione naturale. I biologi hanno studiato i meccanismi dell'evoluzione fin da quando la teoria di Darwin[4] incominciò a riscuotere i primi consensi. Ciò che sorprese maggiormente gli stessi biologi fu la relativa velocità con la quale si sono evolute le forme di vita a partire dai primi fossili per arrivare a strutture molto più complesse quali i mammiferi ed in particolare l'uomo.

Il meccanismo che ha guidato questa evoluzione non è ancora completamente chiaro ma alcune sue peculiarità sono note. Un essere vivente deve la sua struttura

[4] Charles Darwin (1809-1882) biologo inglese padre della teoria dell'evoluzione.

fisica ad un processo di decodifica e di interpretazione dei cromosomi. Il modo in cui ciò avviene è per lo più sconosciuto ma vi sono alcune parti chiare ed universalmente accettate.

L'evoluzione è un processo che opera sui cromosomi e non sugli esseri viventi generati dalla loro decodifica.

La selezione naturale è un collegamento tra i cromosomi e le performance delle loro strutture decodificate. I processi della selezione naturale fanno sì che i cromosomi che generano strutture di successo si riproducano con maggiore frequenza di quelli che generano strutture poco valide.

Il processo della riproduzione è il momento in cui avviene l'evoluzione. Le mutazioni fanno sì che un figlio abbia cromosomi diversi dai suoi genitori biologici ed il processo di ricombinazione crea cromosomi differenti mescolando il materiale genetico dei genitori.

L'evoluzione biologica non ha memoria. Tutto ciò che si sa di un individuo ben inserito nell'ambiente in cui vive è contenuto nel suo patrimonio genetico e sarà perso al momento della sua morte.

Nei primi anni settanta si pensò che questi meccanismi diffusi in natura potessero essere validi per risolvere problemi di elevata complessità anche in ambito informatico. Nacquero così gli algoritmi genetici. Questi algoritmi operano su sequenze di cifre binarie, 0 ed 1; queste sequenze saranno chiamate cromosomi. Come in natura, gli algoritmi genetici risolvono il problema di trovare dei cromosomi efficienti manipolando ciecamente il materiale genetico. Inoltre questi algoritmi non hanno alcuna conoscenza del problema da risolvere. Le uniche informazioni sono date da una valutazione del singolo cromosoma prodotto e questa informazione altera solamente la possibilità di riproduzione degli individui agevolando quelli ritenuti migliori.

3.1.2 Uno sguardo generico agli algoritmi genetici

L'astrarre un fenomeno naturale in un algoritmo genetico presenta una difficoltà: è necessario collegare l'algoritmo al problema da risolvere. Bisogna trovare un modo per codificare le soluzioni del problema nei rispettivi cromosomi e trovare una funzione di valutazione che restituisca un valore della qualità di una soluzione a partire dai cromosomi che la codificano.

Le tecniche per codificare le soluzioni possono variare da problema a problema e da algoritmo ad algoritmo; la più usata prevede la codifica tramite stringhe di bit. Probabilmente non esiste una codifica che funzioni meglio delle altre in ogni problema e nella sua scelta è richiesta una certa dose di esperienza e di mestiere.

La funzione di valutazione, normalmente chiamata con il termine "fitness", è il vero trait d'union tra il problema reale e l'algoritmo risolutivo. Dato un cromosoma, la "fitness" ci dice quanto l'individuo che il cromosoma genera sia in grado di risolvere il problema. In natura questo ruolo è svolto dall'ambiente circostante che premia gli individui che meglio vi si adattano; ad esempio un orso polare avrà una pessima valutazione se l'ambiente in questione è l'equatore mentre sarà premiato se l'ambiente in cui opera è il polo. Partendo da queste considerazioni è possibile creare un algoritmo genetico. Se l'algoritmo funziona si otterranno delle soluzioni altamente specializzate e valide pur partendo da una popolazione iniziale scadente purché vi sia una certa dose di variabilità.

Cerchiamo ora di capire come opera praticamente un algoritmo genetico. Il suo funzionamento si può schematizzare in sei passi:

1. Crea una popolazione iniziale di cromosomi (individui).
2. Valuta singolarmente ogni cromosoma della popolazione
3. Crea nuovi cromosomi combinando i cromosomi già esistenti; applica inoltre eventuali variazioni e ricombinazioni.
4. Valuta i nuovi cromosomi ed si inserisce nella popolazione.
5. Elimina gli individui in sovrannumero.

6. Se si è raggiunto un risultato valido o è scaduto il tempo a disposizione ritorna la popolazione attuale altrimenti si riprende dal punto 3.

Ogni singolo passo presenta delle difficoltà di realizzazione che possono variare da problema a problema.

La difficoltà di realizzare una "fitness", un ambiente di vita, sono così correlate al singolo problema che è impossibile farne una trattazione generale. È invece possibile citare le metodologie più usate per selezionare e per incrociare due o più cromosomi. Queste tecniche devono garantire che un individuo valido abbia maggiori possibilità di riprodursi di un individuo scadente.

Le due tecniche più utilizzate nella scelta dei "genitori" sono la tecnica detta "roulette" e la tecnica del "torneo".

La prima prevede che ad ogni individuo sia assegnato un settore di cerchio proporzionale al suo valore e che il valore totale di tutti gli individui sia pari al cerchio intero. A questo punto la roulette gira e il cromosoma estratto sarà uno dei genitori. In definitiva si ha che la probabilità di essere estratti è proporzionale al valore della propria fitness.

Nel secondo metodo, il "torneo", le probabilità di estrazione sono pari per tutti gli individui. Vengono estratti due o più individui e viene scelto quello che tra questi ha il valore maggiore. Scelti i genitori bisogna procedere all'accoppiamento.

3.2 La simulazione

In ogni attività di ricerca e di sviluppo, la simulazione costituisce un momento importante. Simulare il comportamento di un oggetto può infatti portare alla conoscenza di pregi e difetti non sospettabili o comunque può confermare le ipotesi formulate. Spesso le simulazioni richiedono grandi sforzi per essere portate a termine. Questi sforzi possono essere economici come nel caso di un crash test di una Ferrari oppure possono essere sforzi di tipo temporale. È questo il caso delle simulazioni al computer che richiedono un notevole impiego di tempo per essere portate a termine. Le simulazioni, quindi, sono per loro natura eventi da ripetere il numero di volte strettamente necessario e i loro risultati sono assai importanti. Le modalità con le quali una simulazione avviene possono variare con il variare delle necessità, della tecnologia e dell'evento da simulare. Nel nostro caso, l'oggetto da simulare è del codice VHDL. In realtà ciò che si simula è un dispositivo elettronico che per facilità viene rappresentato mediante una sua descrizione in linguaggio VHDL. In questo modo si possono avere simulazioni al computer senza la necessità dell'oggetto fisico; questo permette di avere conoscenze importanti nelle prime fasi di sviluppo e quindi avere la possibilità di sfruttarle al meglio.

Il linguaggio VHDL è un insieme di istruzioni formulate mediante una sintassi convenzionale e in una forma grammaticalmente corretta. Questa sintassi è stata pensata per mantenere una certa corrispondenza tra la realtà fisica dell'oggetto e la sua descrizione. Per simulare il VHDL è necessario l'utilizzo di appositi programmi; questi programmi sono detti simulatori VHDL. Il loro compito è quello di ricevere una descrizione di un circuito e di renderla eseguibile dal calcolatore. Per far questo è possibile seguire due strade: la compilazione del sorgente o l'interpretazione. La compilazione prevede che il computer, ricevuto un listato in VHDL, lo elabori nel suo complesso e ne fornisca in uscita una versione in codice binario eseguibile. L'utilizzatore dovrà poi fare eseguire dal computer questa versione così tradotta. Ciò che avviene in pratica è la trasformazione del

sorgente in VHDL nel suo equivalente in linguaggio C o direttamente in linguaggio macchina. L'interpretazione, invece, prevede che ogni istruzione in VHDL venga tradotta singolarmente in una forma comprensibile all'elaboratore elettronico e che venga immediatamente eseguita. In un certo senso è equiparabile ad un lavoro di traduzione simultanea. Questi approcci sono entrambi validi anche se risulta preferito il metodo della compilazione. I motivi sono da ricercarsi soprattutto nel minore tempo richiesto per l'esecuzione di più simulazioni. Infatti, se devo eseguire più di una simulazione, il simulatore che interpreta il codice deve effettuare ogni volta una traduzione perdendo così del tempo. La versione compilata, invece, sarà ancora valida e non deve più sprecare tempo a ritradurla. Molti simulatori commerciali utilizzano il metodo della compilazione.

Il simulatore da noi usato è del tipo compilativo. Il codice VHDL viene infatti letto e tradotto nell'equivalente C istruzione per istruzione. In questo modo si ha una corrispondenza uno ad uno tra istruzione VHDL ed istruzione C ed il codice generato non risulta ottimizzato. Alla fine ciò che si ottiene è un sorgente in linguaggio C. Assieme ad esso vengono generati altri files che contengono informazioni utili per il simulatore. Il codice C così ottenuto viene poi aggregato al codice del simulatore ed il tutto viene compilato da un compilatore standard. Quello che si ottiene è una struttura compatta che riunisce in sé il circuito da simulare ed il simulatore stesso.

Il codice del simulatore e quello del traduttore sono stati modificati per ottenere le informazioni necessarie, in fase di simulazione, per il corretto funzionamento dell'algoritmo genetico da realizzare; ovverosia per creare un ambiente tale da generare un’evoluzione proficua. Inoltre il simulatore è stato integrato con un algoritmo genetico che permettesse la ricerca delle soluzioni migliori al problema della generazione delle sequenze di test.

Le simulazioni da noi effettuate hanno sempre utilizzato questo simulatore ed i risultati ottenuti sono poi stati confrontati con quelli forniti da altri simulatori commerciali.

3.3 Modello di guasto

I modelli di guasto introdotti e analizzati per descrizioni a basso livello non sono immediatamente applicabili e trasferibili a descrizioni ad alto livello. Si presenta cosi la necessita di introdurne di nuovi o di adattare quelli esistenti estendendone la loro definizione.

Un modello di guasto di tipo "*stuck at*" ha certamente un significato reale e ben preciso se si pensa ad un layout di un circuito o a una sua descrizione comportamentale a basso livello. Per rilevare un guasto di questo tipo è necessario fornire agli ingressi del circuito una serie di pattern che riescano ed eccitare il guasto.

Quando, invece, si ha una descrizione ad alto livello sorgono numerosi problemi. Non ha infatti senso parlare di guasti per istruzioni complesse come ad esempio "`a=b*c`"; un istruzione di questo tipo utilizza infatti tre registri ed un moltiplicatore, oggetti composti da un notevole numero di porte logiche, ognuna passibile di guasto!

Per individuare quindi una possibile presenza di guasti bisognerebbe eseguire quest'operazione un elevato numero di volte ed ogni volta con valori differenti nei registri. Questi valori non dovrebbero essere inoltre casuali ma scelti con cura per poter eccitare tutti i possibili guasti.

Un approccio di questo tipo non è certamente accettabile se si pensa alla attuale complessità dei circuiti ed alla dimensione dei loro registri. Occorre pero tenere presente che spesso i registri o i blocchi funzionali di un circuito sono riutilizzati più volte durante l'esecuzione della descrizione ad alto livello. Se una descrizione deve eseguire dieci moltiplicazioni, probabilmente utilizzerà dieci volte lo stesso moltiplicatore; non e più necessario quindi testare completamente il blocco per ogni moltiplicazione perché se cosi si facesse, alla fine, si sarebbe testato dieci volte lo stesso moltiplicatore! Non esiste più una corrispondenza uno ad uno tra guasto e descrizione ma abbiamo un approccio più "distribuito". Eccitare una

istruzione non significa necessariamente eccitare un guasto ed eccitarla più volte può essere una perdita di tempo in quanto i suoi blocchi costitutivi possono essere già stati eccitati altrove.

Il modello di guasto che riteniamo adatto alle descrizioni ad alto livello fa riferimento alla eseguibilità o meno delle istruzioni componenti il codice VHDL. La reiterata attivazione di un'operazione viene considerata come prova di collaudo della porzione di circuito da essa descritto. Qualora l'istruzione non sia raggiungibile, ovvero non si riesca ad eseguirla il numero di volte desiderato, questa viene considerata non testabile.

3.4 L'algoritmo

L'algoritmo proposto, per adempiere al meglio alle necessità precedentemente spiegate, opera in due fasi: pretest e test vero e proprio.

Durante il pretest l'obbiettivo prefissato è quello di eccitare il maggior numero di istruzioni più volte utilizzando il minor numero di pattern possibili. Come ciò avvenga sarà spiegato durante la trattazione dell'algoritmo genetico che ne regola il funzionamento. In questo modo si può avere, in tempi decisamente brevi, una serie di pattern che eccitano i componenti più usati e più sfruttati all'interno del circuito. Se il circuito è relativamente semplice questa simulazione offre risultati validi ed in alcuni casi indistinguibili da quelli ottenibili con un test completo. Inoltre questa presimulazione fornisce una stima sulla facilità di eseguire o meno alcune istruzioni. Questo può essere utile in fase di ottimizzazione. Se una zona del circuito viene eccitata spesso converrà concentrare gli sforzi per ottimizzare quella zona piuttosto che un'altra che viene chiamata in causa magari solo poche volte.

Il test vero e proprio serve invece ad eccitare quelle zone difficilmente raggiungibili e quindi a stanare i guasti in zone più remote e meno utilizzate. L'algoritmo utilizzato è differente in quanto lo scopo non è più eccitare indiscriminatamente un elevato numero di istruzioni ma riuscire ad attivare una o più volte una singola e ben precisa istruzione.

Se il circuito da testare è particolarmente complesso, attivare le funzioni una singola volta può non essere sufficiente. La complessità dei blocchi costitutivi, della logica di controllo e dei bus di trasmissione dei dati nascosta dietro l'innocua apparenza di un'istruzione elementare, può essere tale da richiedere una reiterazione del processo di test per ottenere un buon risultato. A questo scopo è stato introdotto un valore di soglia, modificabile a piacere dall'utente, che indica il numero di volte che un'istruzione deve essere eseguita dallo stesso pattern per venire considerata sufficientemente testata.

Per circuiti di complessità media ed elevata, un valore di reiterazioni valido si e rivelato essere tre, valore che e stato introdotto come default. La modifica di questo valore può influenzare notevolmente il tempo impiegato per ottenere i risultati e la qualità dei medesimi. Un valore di soglia inferiore produrrà risultati in minor tempo ma la loro validità, per verificare la testabilità del circuito, sarà indubbiamente minore. Aumentare il valore di soglia significa invece aumentare il tempo di calcolo senza la certezza di un incremento dei risultati. La modifica di questo parametro va quindi fatta con cautela e solo dopo una certa esperienza nell'uso del programma.

Se durante la presimulazione o durante il test di un preciso segnale si verifica che il medesimo pattern utilizzato eccita un'altra istruzione un numero di volte superiore alla soglia prefissata, questa istruzione sarà considerata come già testata.

Questa tecnica prende il nome di *fault dropping* e permette di ridurre il numero di istruzioni da testare eliminando quelle superflue e riducendo quindi il tempo di calcolo.

L'algoritmo genetico usato da RAGE per trovare i cromosomi adatti alla soluzione dei problemi è piuttosto complesso ed articolato. L'algoritmo è scomponibile in varie parti, ognuna con le proprie caratteristiche e con dei parametri regolabili dall'utilizzatore.

3.4.1 BRAHMA: la creazione

La prima parte dell'algoritmo prevede la creazione di una popolazione iniziale di individui. Un individuo è un'insieme di bit pari ad un multiplo intero del numero di ingressi del circuito. Questo insieme di bit rappresenta una sequenza che sarà applicata alla descrizione in VHDL e che permetterà di eccitare un insieme di istruzioni.

Lo scopo è quello di trovare pattern che eccitino una istruzione ben determinata scelta di volta in volta un numero di volte pari o superiore al valore di soglia. È

possibile agire in questa fase operando su due parametri: il numero di individui costituenti la popolazione e la lunghezza iniziale degli individui.

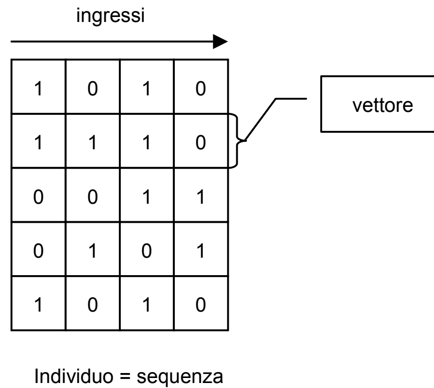

Figura 7: ***Codifica di un individuo o sequenza.***

È necessario prestare particolare attenzione al numero di individui che formano la popolazione; un numero troppo basso porterebbe a dei miglioramenti generazionali lenti e poco significativi rallentando così il processo evolutivo. Un numero troppo elevato potrebbe consentire invece la sopravvivenza di individui poco validi e quindi permettere loro di riprodursi rallentando nuovamente il processo evolutivo. È stato preimpostato un valore di 50 individui che è parso essere un buon compromesso. Al momento della creazione iniziale ogni individuo viene valutato dalla funzione di fitness. Il valore così ottenuto viene legato al cromosoma in quanto strettamente correlato. Gli individui della prima generazione vengono creati in modo assolutamente casuale contrariamente a ciò che avverrà per le future generazioni. Se questi cromosomi decodificano soluzioni valide per la risoluzione del problema, l'algoritmo si arresta e passa al problema successivo, altrimenti inizia la fase genetica vera e propria.

3.4.2 VISNU: l’evoluzione

In questa fase le soluzioni casuali vengono elaborate al fine di essere perfezionate ed affinate. Ciò avviene mediante il processo di riproduzione.

All'interno della popolazione vengono scelti due individui con il criterio del "torneo" (cfr. 3.1 Algoritmi Genetici) che daranno vita ad un nuovo elemento.

L'incrocio dei cromosomi avviene tramite *crossover* uniforme verticale. Immaginando un individuo come una matrice di bit, il figlio sarà composto utilizzando le colonne del padre e quelle della madre in maniera euristica. La prima colonna sarà scelta a caso tra le due dei genitori, così la seconda e la terza e via via sino all'esaurimento delle colonne. Poiché i genitori possono essere di lunghezza differente, le colonne del genitore più corto saranno completate con valori casuali fino alla lunghezza del genitore più lungo.

Questo avviene solo durante l'accoppiamento ed alla fine di questo i genitori riprendono le loro sembianze originali. In questo modo il figlio avrà la stessa lunghezza del più lungo dei genitori.

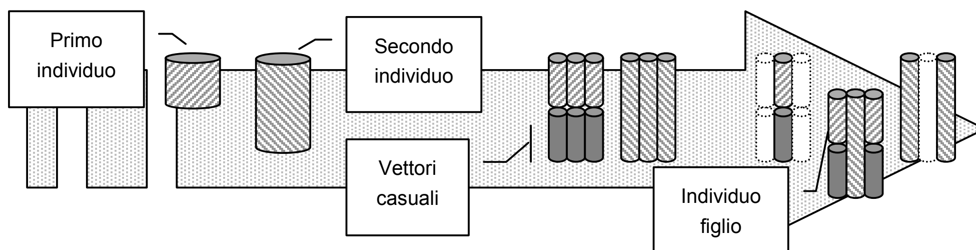

Figura 8: ***Schema di funzionamento del crossover verticale.***

Questo tipo di *crossover* offre ottimi risultati se l'individuo è composto da un elevato numero di colonne, ovvero se il circuito ha un elevato numero di ingressi. Nel caso in cui il circuito avesse un solo ingresso, il figlio sarebbe l'esatta copia di uno dei genitori; se gli ingressi fossero due, ciò accadrebbe statisticamente ogni due riproduzioni. In questo modo non si avrebbe alcun processo evolutivo.

Per questo motivo è stato inserito un secondo tipo di *crossover* attivabile dall'utente che prende il nome di *crossover* uniforme orizzontale ed opera in maniera speculare rispetto al precedente. Infatti invece di usare le colonne come materiale genetico da incrociare utilizza le righe.

Quando attivato, questo *crossover* agisce con una frequenza pari al reciproco del numero di ingressi del circuito; le restanti volte viene applicato il *crossover* verticale. In questo modo se si ha un solo ingresso viene applicato sempre il x-over orizzontale; se gli ingressi sono due (quindi due colonne) sarà applicato una volta ogni due e così via.

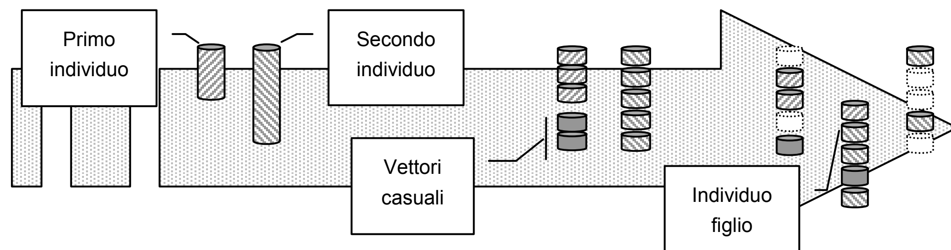

Figura 9: ***Schema di funzionamento del crossover orizzontale.***

Dopo aver costruito un nuovo individuo si procede ad una sua eventuale mutazione. La probabilità che un nuovo individuo subisca una mutazione è regolata dall'utente. Normalmente viene mutato un individuo ogni dieci ma nulla vieta di variare questo valore.

Aumentando il numero di individui mutati si corre il rischio di destabilizzare troppo la popolazione introducendo un eccessivo quantitativo di materiale genetico nuovo. Va ricordato comunque che un certo fattore di mutazione è sempre necessario per rivitalizzare una popolazione che altrimenti tenderebbe ad impauperarsi ed a livellarsi. Una mutazione crea una novità; troppe novità possono essere dannose, troppo poche possono rendere sterile la popolazione.

Le mutazioni possono avvenire in tre modi: aggiunta, sottrazione e sostituzione.

La mutazione aggiuntiva allunga semplicemente il cromosoma aggiungendo una riga casuale al suo interno; in questo modo il cromosoma si allunga. La posizione in cui questa riga viene aggiunta è casuale.

La mutazione sottrattiva, invece, elimina casualmente una riga rendendo cosi il cromosoma più corto. Questi sono i motivi per cui a partire da una popolazione di individui di eguale lunghezza è possibile ottenere individui di lunghezza differente.

La terza mutazione, invece, scambia una riga del cromosoma con un'altra generata casualmente. Questa mutazione non modifica ovviamente la lunghezza dell'individuo.

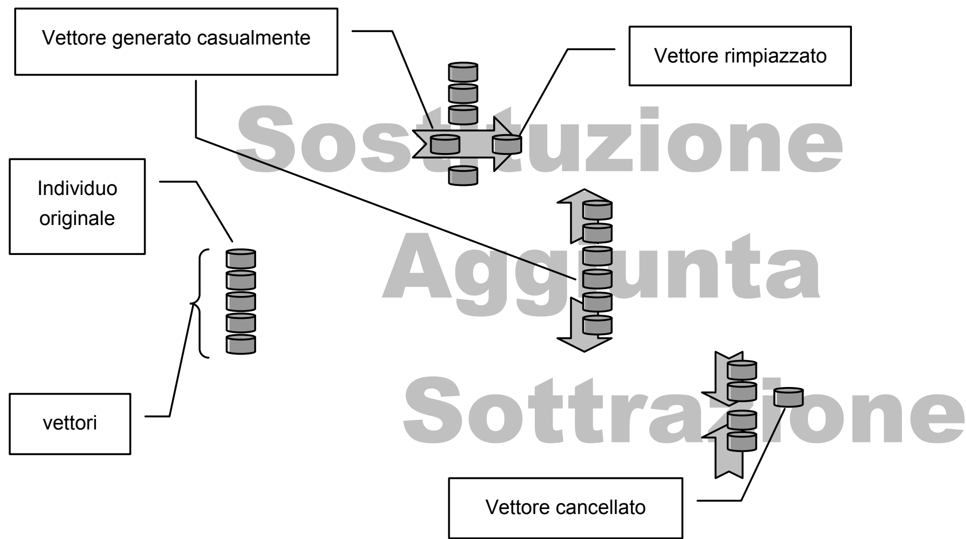

Figura 10: ***Operazioni di mutazione.***

La scelta tra queste mutazioni è casuale.

Il nuovo cromosoma a questo punto è completato e deve quindi essere valutato con la procedura di fitness. Dopo essere stato valutato, il nuovo individuo viene inserito nella popolazione. Il numero di nuovi individui che viene creato ad ogni iterazione è regolabile dall'utente. Generalmente il numero di individui nuovi è uguale od inferiore a quello della popolazione ma nulla impedisce il contrario. Un elevato numero di nuovi individui può però rallentare il programma senza accelerare il processo evolutivo. Infatti continueranno ad incrociarsi sempre i medesimi individui e non quelli appena creati che per riprodursi devono attendere una nuova iterazione.

3.4.3 SHIVA: la selezione

A questo punto la popolazione si trova ad essere in sovrannumero. Infatti coesistono tutti i vecchi individui più quelli creati recentemente. Come avviene in natura un certo numero di individui deve morire.

Bisogna scegliere quali individui devono morire e quali vivere. Le soluzioni possibili sono due: l'eliminazione dei cromosomi "vecchi" oppure di quelli meno validi. L'eliminazione dei meno validi è stata la soluzione adottata in quanto era quella che portava a risultati migliori. La popolazione viene quindi ridotta al numero stabilito di individui.

A questo punto il ciclo riprende con la creazione di una nuova generazione e così via.

3.4.4 Casi particolari

Se dopo un certo numero di iterazioni la popolazione non migliora, ovvero se il miglior individuo ha sempre lo stesso valore, l'algoritmo genetico si modifica leggermente. Viene aumentato il fattore di mutazione e, se attivato, entra in funzione il block x-over.

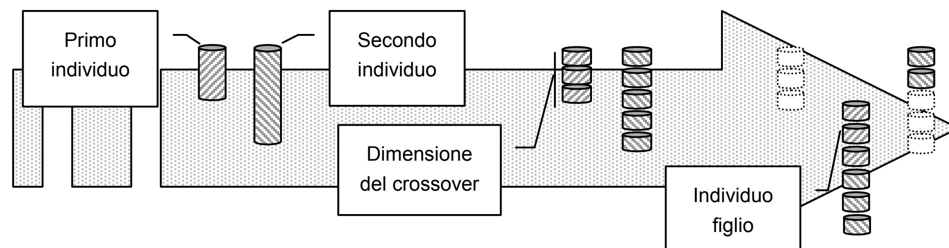

Figura 11: ***Schema di funzionamento del block crossover con valore 2.***

Questo nuovo tipo di *crossover* sostituisce i precedenti e serve per allungare le sequenze sperando in un loro miglioramento. Scelto un individuo, viene calcolata la nuova lunghezza del figlio come la lunghezza del padre scelto moltiplicata per il valore assegnato al block x-over. Un valore tipico può essere compreso tra 1,2 e 2. Calcolata la lunghezza del figlio, la prima metà dell'individuo viene creata usando le prime righe del padre e la seconda le ultime della madre. Qualora non si

abbia un numero sufficiente di righe a disposizione, queste saranno create a caso. Se si ha un miglioramento della popolazione, l'algoritmo genetico riprende il suo funzionamento normale altrimenti prosegue in questo modo per un numero determinato di iterazioni.

La ricerca di soluzioni per un determinato problema si ritiene esaurita quando viene individuata una soluzione valida oppure quando si è raggiunto un limite prefissato nel numero di iterazioni. Qualora si sia giunti ad una soluzione valida, questa viene scritta su un file.

3.4.5 ARMAGEDDON: la reincarnazione

Esaurita la ricerca di una soluzione per un problema, si passa al problema successivo. La popolazione viene azzerata e ne viene creata una nuova. Azzerare completamente una popolazione significa perdere completamente tutto il processo evolutivo avvenuto precedentemente.

Questo ovviamente non ha alcuna importanza se i problemi da risolvere sono completamente scorrelati tra loro. Nel nostro caso, poiché il problema è l'attivazione di una istruzione ed il problema successivo è l'attivazione di una ulteriore istruzione che normalmente è relativamente vicina alla precedente, si può ipotizzare una correlazione tra i due problemi. Per non perdere quindi quanto di buono era già stato fatto precedentemente, si è introdotto un coefficiente di "rinascita", il *reborn rate.* Questo coefficiente indica il numero di individui che sopravvivono ad un problema e vengono riutilizzati per il successivo. Questi individui vengono inseriti nella nuova popolazione dopo essere stati rivalutati con la fitness tarata per il nuovo problema. La nuova popolazione viene poi completata con nuovi individui creati a caso.

A questo punto il ciclo riparte dall'inizio fino all'esaurimento dei problemi.

3.5 Implementazione

L'algoritmo è stato implementato in C modificando opportunamente il codice di un simulatore VHDL, sviluppato all'Università di Pittsburgh, in modo da inserire al suo interno il genetico.

Il simulatore in questione è composto da un compilatore che riceve in ingresso una descrizione VHDL e, dopo aver verificato che sia valida, crea una sua rappresentazione interna e un database contenente l'albero delle dichiarazioni di assegnamento.

La rappresentazione generata viene poi compilata come fosse un sorgente C ed unita alla libreria che contiene il simulatore vero e proprio, ottenendo un eseguibile che, sfruttando il database, permette la simulazione

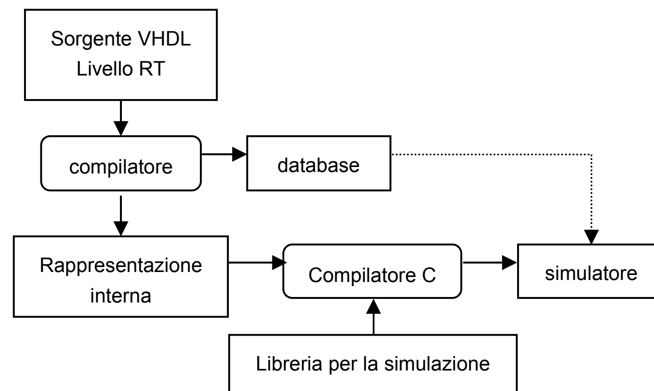

Figura 12: ***Creazione del simulatore.***

In un primo tempo si è modificato il compilatore così da ottenere maggiori informazioni sulla descrizione VHDL. In particolare si è fatto sì che la rappresentazione creata contenesse oltre alle istruzioni il loro numero di riga, il nome della procedura di cui facevano parte, i nomi dei segnali che coinvolgevano e se quest'ultimi venivano scritti o letti. Tutto ciò permette l'adozione del modello di guasto precedentemente introdotto (cfr. 3.3 Modello di guasto).

Di conseguenza è stato necessario intervenire sul sorgente del simulatore in modo da differenziare le procedure di assegnazione "normali" da quelle che coinvolgono le istruzioni maggiormente dettagliate, cioè quelle istruzioni di cui si conosce la collocazione nel codice, la procedura di appartenenza e la modalità con cui agiscono sui segnali.

In questo modo è stato creato uno strumento in grado di simulare una descrizione VHDL tracciando le varie istruzioni così da permetterci di sapere quando, dove e quante volte un segnale viene scritto o letto. Queste informazioni permettono inoltre di assegnare una percentuale di copertura alle sequenze introdotte nella simulazione, così da valutare quale sia la migliore.

Sulla base delle nuove informazioni ottenute dalla descrizione VHDL attraverso il compilatore modificato, si è messa mano all'implementazione del genetico vero e proprio. L'interfaccia utente del simulatore è stata rimossa e sostituita con le tre procedure di creazione, selezione e rimozione degli individui: brahma, shiva e visnu.

Processo	Lista Operazioni
08 D <= E; 09 end process; 10 P: process 11 begin 12 wait until ck='1'; 13 A <= B ; 14 if A < C then 15 B <= B+1; 16 else 17 B <= A ; 18 end if; 19 end process; 10 Q: process 11 begin	segnale 1 B read in line 13 in process P segnale 2 A write in line 13 in process P segnale 3 A read in line 14 in process P segnale 4 C read in line 14 in process P segnale 5 B read in line 15 in process P segnale 6 B write in line 15 in process P segnale 7 A read in line 17 in process P segnale 8 B write in line 17 in process P

Tabella 1: ***Esempio di lista delle operazioni.***

Dalle prime prove è risultato subito evidente come, nonostante il buon funzionamento dell'algoritmo, una fitness basata solo sul numero di istruzioni

eseguite fosse insufficiente. Inoltre per descrizioni complesse, contenenti cicli o scelte, risultava più proficuo attivare singolarmente i vari segnali.

Alla luce di queste considerazioni si è messo nuovamente mano al compilatore, onde creare un file contenente la lista dei segnali che si intende testare. Inoltre usando il parser VHDL commerciale LEDA per l'analisi del *control-data flow* si è creata una matrice di dipendenza fra le varie istruzioni. In questa matrice sono contenute le probabilità congiunte dei vari segnali, ovverosia la probabilità che un dato segnale sia attivato qualora un altro lo sia già stato.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	-	1	1	1	0.5	0.5	0.5	0.5
2	0.5	-	1	1	0.5	0.5	0.5	0.5
3	0.5	0.5	-	1	0.5	0.5	0.5	0.5
4	0.5	0.5	1	-	0.5	0.5	0.5	0.5
5	0.5	0.5	0.5	0.5	-	1	0.25	0.25
6	0.5	0.5	0.5	0.5	0.25	-	0.25	0.25
7	0.5	0.5	0.5	0.5	0.25	0.25	-	1
8	0.5	0.5	0.5	0.5	0.25	0.25	0.25	-

Tabella 2: ***Esempio di matrice di adiacenze per l'esempio in Tab. 1.***

In questo modo si può costruire una fitness migliore, in quanto col metodo precedente era premiato il pattern che attivava il maggior numero di segnali, ora invece viene premiato quello che più si avvicina al segnale che vogliamo attivare, cioè quello che attiva i segnali a lui adiacenti.

Una volta stabilita l'esistenza di una "relazione" si possono fare alcune considerazioni per affinare ulteriormente l'algoritmo.

La prima considerazione riguarda la soglia, in base al numero di volte che si vuole attivare un segnale si stabilisce il grado di libertà dei suoi segnali adiacenti, saturando la sua funzione di merito in base al coefficiente della matrice di adiacenze che li lega. Quindi se si vuole attivare il segnale 5 tre volte occorre

scegliere preferibilmente quelle sequenze che attivano il segnale 1 sino a sei volte prima degli altri.

La seconda considerazione riguarda la lista dei segnali, come si può facilmente vedere la matrice di adiacenze è ridondante in quanto i segnali che si trovano sulla stessa linea del sorgente VHDL hanno probabilità identica fra loro, infatti se ne viene attivato uno vengono attivati anche gli altri. Di conseguenza è meno oneroso tenere conto delle linee anziché dei segnali che vi appaiono.

Un'altra considerazione riguarda il numero massimo di generazioni; normalmente un algoritmo genetico ha un limite di generazioni prefissato raggiunto il quale passa al segnale successivo. Facendo in questo modo però non si sfrutta a pieno la potenzialità del genetico in quanto esistono descrizioni con segnali posizionati in parti del codice molto difficili da raggiungere anche esse testabili.

Nella realizzazione di RAGE si è invece preferito stabilire un intervallo minimo di variazioni della fitness quale condizione di terminazione. Una volta che il migliore individuo della popolazione ha un merito compreso in questo intervallo per un certo numero di generazioni, vengono cambiati i valori di mutazione e se non migliora si passa al segnale successivo. Con questo metodo si riescono a raggiungere i massimi della funzione di fitness senza rimanere ingannati da eventuali punti di sella.

L’ultima considerazione riguarda i segnali che fanno parte delle istruzioni eseguite in fase di reset. Questi segnali, poiché la simulazione avviene sempre con il segnale di reset disattivo, non vengono mai attivati, occorre quindi che il genetico se ne accorga in modo da non sprecare tempo di calcolo nel cercare di testarli.

Tenendo conto di queste osservazioni nell’algoritmo si possono identificare tre fasi distinte.

- Presimulazione: l’algoritmo genetico è spinto a generare sequenze che coprano il maggior parte delle istruzioni. In questo caso la fitness è data

semplicemente dal numero di operazioni attivate, tenendo conto della soglia. In questa fase la maggior parte delle operazioni di “facile” copertura vengono testate con una sola sequenza. La presimulazione contribuisce solitamente al 50%-70% della *fault coverage* totale.

- Simulazione: viene selezionata un’operazione e l’algoritmo genetico cerca di attivarla un numero di volte pari alla soglia. In questa fase la fitness è quella descritta in precedenza e dipende dalla matrice di adiacenza.
- Eliminazione delle operazioni già coperte: dopo aver trovato una sequenza che attivi una data operazione si eliminano dalla lista tutte quelle che vengono coperte dalla stessa sequenza. Più dell’80% delle operazioni che non sono state coperte in presimulazione vengono eliminate in questa fase.

Le sequenze generate dall’ATPG così creato sono state poi applicate mediante un *fault simulator* alla lista di guasti *stuck-at* dell’implementazione a livello *gate* sintetizzata mediante Synopsys ottenendo così una percentuale della copertura dei guasti (*fault- coverage*).

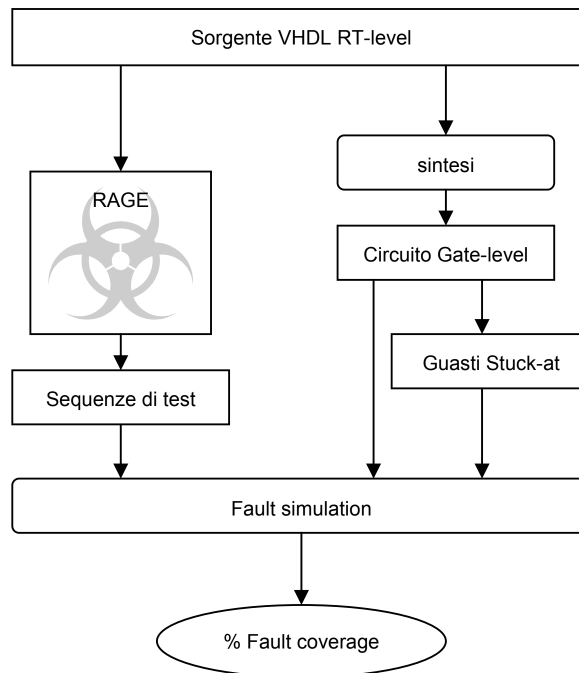

Figura 13: ***Flusso di valutazione.***

I primi test danno subito conferma della bontà dell'algoritmo, infatti si ottengono buone coperture, spesso migliori di ATPG che lavorano a livello *gate*, ma appaiano anche dei difetti.

Il primo e più evidente dipende dal simulatore adottato, ed in particolare dalla cattiva gestione della memoria che impedisce la simulazione di descrizioni troppo complesse. In realtà le perdite di memoria sono minime ma poiché il genetico compie molte simulazioni ogni qual volta calcola la fitness alla fine la memoria allocata risulta talmente tanta da mettere in crisi il sistema operativo.

Questo problema viene risolto utilizzando una libreria di gestione della memoria munita *automatic garbage collector*, che si occupa di minimizzare l'allocazione della memoria durante l'esecuzione. Inoltre sono ottimizzati sia i motori di ricerca nelle liste introducendo delle tavole di *rehash* sia la gestione della matrice di adiacenze. Così facendo il rallentamento introdotto dal garbage collettor risulta minimo nei confronti di quello precedentemente causato dal sistema operativo accendendo al disco, per sopperire alla mancanza di memoria, e delle ottimizzazioni dei sistemi di ricerca e archiviazione dei dati.

L'altro problema viene introdotto dalla matrice di adiacenze, la quale risulta inutile se non dannosa nelle descrizioni contenenti *case* annidati poiché assegna una probabilità uguale a ciascuno dei rami, così che la funzione di fitness valuta ugualmente validi individui che non lo sono. Questo fatto fa sì che il genetico riesca a testare le istruzioni all'interno dei case con molta difficoltà e spesso per puro caso.

Per risolvere questo problema è stato necessario implementare un sistema di correzione della matrice delle adiacenze. Durante il funzionamento del programma, quindi, i valori della matrice vengono cambiati in base a come si evolvono gli individui, cercando di assegnare dei coefficienti di demerito a quelle parti della descrizione che in un primo tempo sembravano portare all'attivazione del segnale desiderato ma che poi si sono rilevate infruttuose.

Purtroppo questo metodo non ha dato i frutti sperati, infatti i valori della matrice di adiacenze venivano alterati in modo impercettibile, quindi è stato necessario introdurre un vero e proprio sistema di apprendimento. Invece di utilizzare la matrice fornita dal LEDA all'inizio della presimulazione se ne crea una formata da valori nulli, durante la presimulazione e la simulazione si inseriscono le probabilità di esecuzione delle varie istruzioni che vengono sfruttati nella calcolo della funzione di fitness.

In particolare ogni qual volta viene eseguita una simulazione si calcola la probabilità di esecuzione di ogni singola istruzione e la si media con quella già presente nella matrice. La media dei due valori tiene conto del numero di sequenze introdotte. Se l'istruzione non viene eseguita in una sequenza di test, quindi la sua probabilità risulta nulla, non viene mediata.

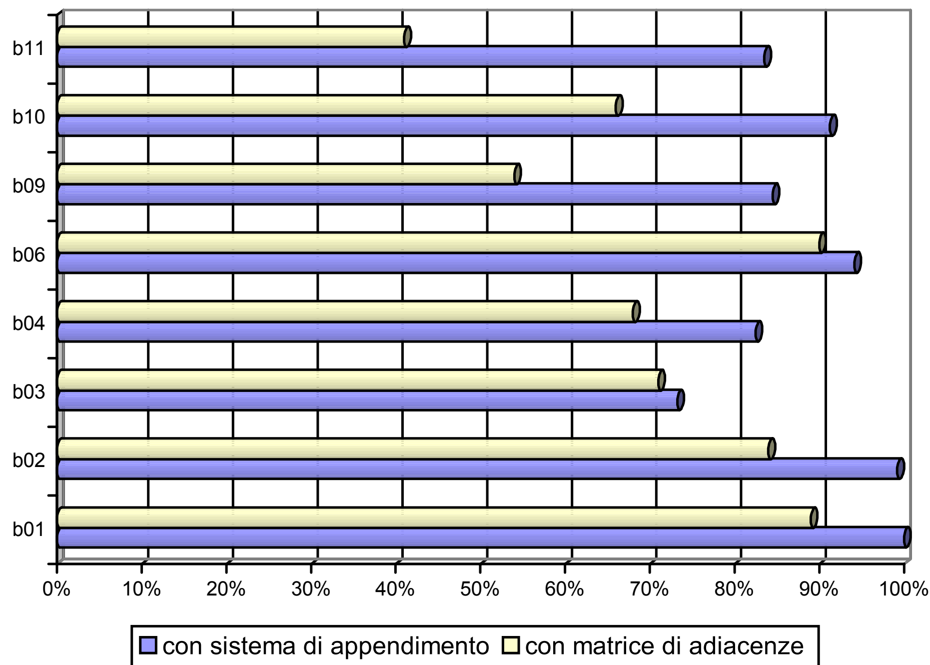

Figura 14: ***Confronto fra i risultati ottenuti con le due diverse fitness.***

Con questo metodo si sfrutta la presimulazione per avere una matrice base che risulta di gran lunga migliore di una ottenuta generando sequenze a caso perché il

genetico cercando di attivare i segnali che non sono stati già attivati fornisce una probabilità anche se bassa a quei segnali che non verrebbero raggiunti casualmente. Inoltre nel processo di simulazione il genetico "impara" anche dagli errori, infatti anche gli individui che non attivano il segnale voluto forniscono delle informazioni utili all'attivazione di altri segnali.

Il sistema di apprendimento è risultato eccellente per aumentare la *fault coverage* delle descrizioni "logicamente" più complesse ed ha permesso a RAGE di svolgere oltre al ruolo dell'ATPG quello di strumento di confronto fra descrizioni supposte equivalenti.

L'ultima miglioria apportata all'implementazione è quella di aggiungere informazioni sul flusso dei dati all'interno della descrizione dedotte dal datebase prodotta dalla compilazione. Queste nozioni aggiuntive permetto al genetico, una volta attivato un segnale, di portarlo all'uscita del circuito, fornendo così oltre all'analisi della testabilità quella dell'osservabilità.

4 Applicazioni di RAGE

4.1 Analisi di testabilità

Come visto nei precedenti capitoli, il collaudo di un circuito non è mai esaustivo. Occorre cioè arrestarsi quando si sia raggiunta una percentuale di rilevazione dei guasti sufficiente. Il livello di sufficienza è puramente soggettivo e dipende dal circuito, dalla qualità che si vuole ottenere, dalle possibilità economiche e da altri fattori variabili di volta in volta. Talvolta non è possibile raggiungere la soglia desiderata ed in questi casi è necessario ricorrere a modifiche strutturali spesso onerose e difficoltose.

Il numero di guasti coperti prende il nome di fault coverage ed è espresso come valore percentuale. È infatti il rapporto tra i guasti coperti e la totalità dei guasti nella fault list.

Poiché non è possibile stilare fault list per descrizioni ad alto livello, la fault coverage deve essere calcolata in maniera differente.

4.1.1 Come usare RAGE per l’analisi di testabilità

L’algoritmo utilizzato da RAGE per calcolare la fault coverage è più complesso e parte da principi completamente differenti. Infatti non esiste più un semplice rapporto tra guasti da coprire e guasti coperti ma tra istruzioni eseguibili, istruzioni eseguite e numero di volte che le istruzioni sono state eseguite.

Data un'istruzione, si suppone che questa debba essere eseguita almeno un certo numero di volte, questo per avvicinarci alla fault coverage. Qualora l' istruzione fosse eseguita un numero maggiore di volte di quello richiesto, le si assegnerebbe una percentuale di copertura pari al 100%.

Qualora fosse eseguita un numero inferiore di volte, le verrebbe assegnata una copertura pari al numero di volte eseguita diviso il numero di volte richiesto.

Supponendo che l'istruzione sia considerate testata quando viene eseguita tre volte, se al termine della simulazione risulta attivata solo due volte, le sarebbe assegnata una percentuale di testabilità del 66%.

La percentuale di testabilità del circuito è la media delle percentuali di testabilità delle singole istruzioni. In questo modo si calcola una stima della fault coverage basandosi su una lista di operazioni e non su una lista di guasti. Questo metodo può sembrare impreciso ed empirico ma propone delle buone concordanze con le fault coverage calcolate a livello gate. Infatti se un istruzione viene eseguita un numero elevato di volte nell'esecuzione dei pattern, anche la porzione di circuito che questa sottende sara largamente utilizzata e quindi testata. Il numero di volte che un'istruzione deve essere testata può variare da circuito a circuito ed è posta a tre come default poiché questo valore è risultato decisamente efficace.

Per usare RAGE come test analizer è necessario fornire al simulatore un file contenente le sequenze da applicare.

Sequenza di test	Significato
#	RESET
101001010010	1° sequenza
010100100100	2° sequenza
101010101010	3° sequenza
#	
100101111111	
100001000100	

Figura 15: ***Esempio di sequenza di test.***

Le sequenze devono essere separate dal carattere # che rappresenta l'esecuzione di un comando di reset. I bit componenti una singola sequenza del

pattern devono seguire l'ordine dei segnali cosi come sono elencati nel file di segnali generato in fase di compilazione del VHDL.

4.1.2 Risultati ottenuti usando RAGE per l'analisi di testabilità

Per verificare l'affidabilità di RAGE come analizzatore di testabilità si sono sintetizzati una serie di circuiti di prova, *benchmarks*, descritti in VHDL a livello RT, le cui caratteristiche sono riportate nella Tabella 3. Questi circuiti sono di differente lunghezza e complessità, essi spaziano dal semplice filtro alla più complessa *fetch* di un microprocessore oppure alla sua unità di esecuzione.

	RT-level		gate-level				
Nome	**Linee VHDL**	**Operazioni**	**Porte**	**FF**	**PI**	**PO**	**Faults**
b01	110	70	45	5	4	2	268
b02	70	25	25	4	3	1	128
b03	141	100	150	30	6	4	822
b04	80	63	480	66	13	8	2640
b06	128	60	66	9	4	6	342
b09	103	64	131	28	3	1	736
b10	167	126	172	17	13	6	952
b11	110	70	366	30	9	6	2148

Tabella 3: ***Circuiti di prova descritti a livello RT.***

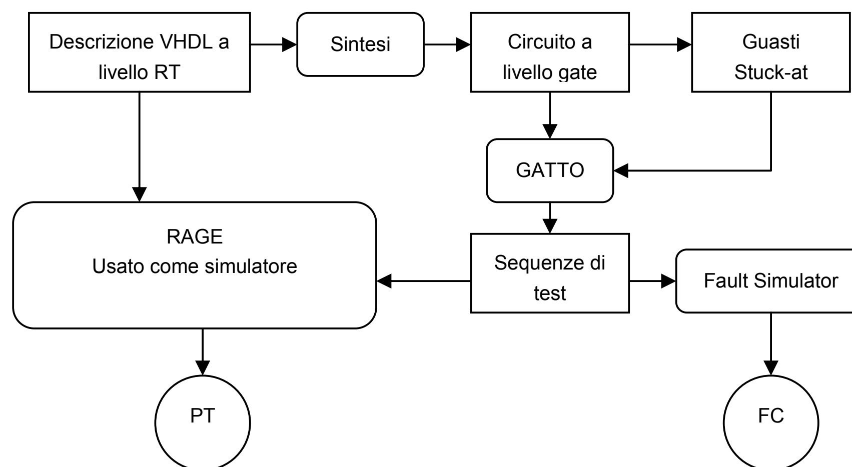

Figura 16: ***Flusso di valutazione di RAGE nell'analisi di testabilità.***

Le descrizioni sono state sintetizzate mediante Synopsys, avvalendosi di una semplice libreria composta da porte logiche elementari. La lista dei guasti *stuck-at* è stata generata utilizzando l'ambiente SUNRISE e mediante un ATPG sviluppato dal Politecnico di Torino e basato sull'utilizzo di algoritmi genetici, ATPG dal nome GATTO, si sono generate alcune sequenze di test.

Le sequenze trovate sono state infine simulate utilizzando RAGE, applicato alla descrizione a livello RT, così da vedere di quanto la percentuale di testabilità, si discosti dalla *fault coverage* ottenuta attraverso un *fault simulator*.

	b01	*b02*	*b03*	*b04*	*b06*	*b09*	*b10*	*b11*
Fault Coverage	100%	99%	73%	83%	94%	85%	91%	84%
Testabilità	100%	99%	75%	76%	95%	86%	89%	92%
Differenza	0%	0%	2%	7%	1%	1%	2%	8%

Tabella 4: Confronto fra l'operation coverage fornita da RAGE e la fault coverage.

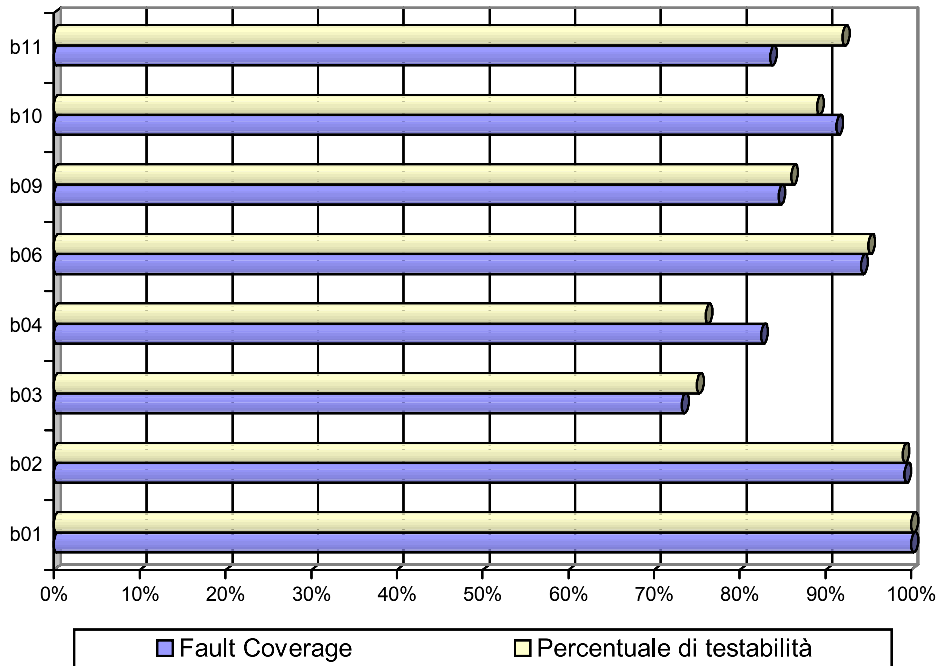

Figura 17: ***Confronto fra l'operation coverage fornita da RAGE e la fault coverage*.**

Come si può vedere, sia nel grafico riportato nella Figura 17 sia dai dati presenti nella Tabella 4 le due coperture sono molto simili tra loro per tutti i

circuiti di esempio, ciò dimostra sia l'utilità di RAGE come simulatore sia una buona corrispondenza tra le operazioni da eseguire e i guasti da coprire.

In particolare vale la pena soffermarsi su due esempi, il b04 e il b11, per cui la descrizione a livello RT produce, una volta sintetizzata, un numero molto alto di porte e di conseguenza un notevole numero di guasti. Questo fatto si può imputare ad una non buona ottimizzazione del layout oppure ad una notevole complessità delle operazioni presenti nella descrizione.

	b01	*b02*	*b03*	*b04*	*b06*	*b09*	*b10*	*b11*
Guasti	268	128	822	2640	342	736	952	2148
Operazioni	70	25	100	63	60	64	126	70
Differenziale	1:4	1:5	1:8	1:42	1:6	1:12	1:8	1:30

Tabella 5: Rapporto tra linee di descrizione e porte sinstetizzate.

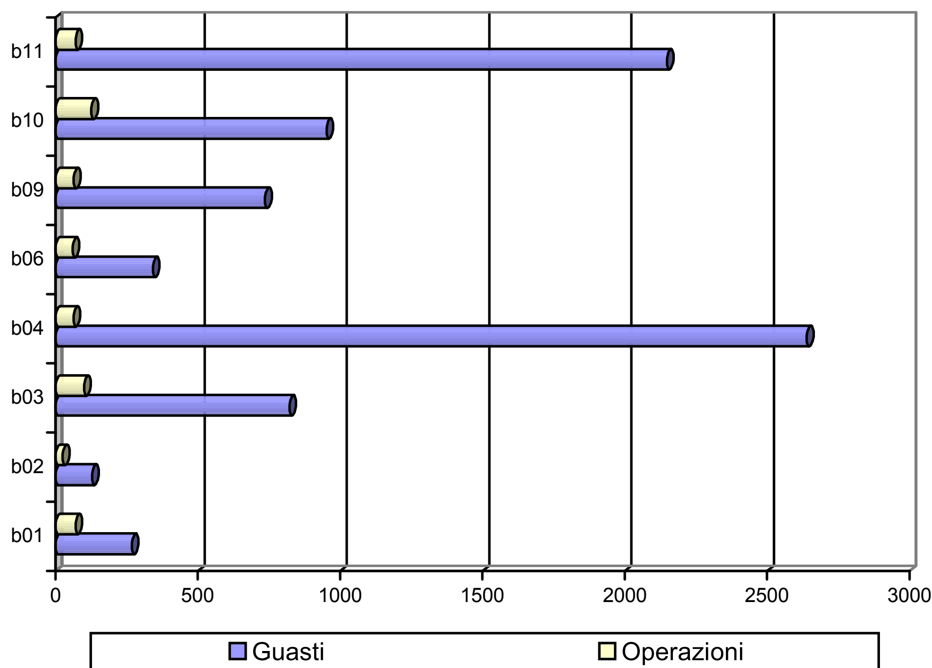

Figura 18: ***Rapporto tra linee di descrizione e porte sinstetizzate.***

Qualsiasi sia la ragione per la quale il circuito sintetizzato presenti un così grande numero di porte, rimane il fatto che una tale differenza tra numero di

operazioni e numero di guasti a livello *gate* potrebbe mettere in crisi un simulatore che lavora ad alto livello come RAGE.

Si osserva, infatti, che la maggior parte delle descrizioni genera circuiti con un rapporto tra guasti e operazioni al si sotto della decina, mentre i due esempi in questione presentano un rapporto superiore a trenta. Ciò non di meno la stima fornita da RAGE con l'*operation coverage* non si discosta mai dalla *fault coverage* finale più del 10%.

Si può quindi affermare che RAGE svolge ottimamente la sua funzione di strumento CAD fornendo al progettista prima della sintesi una buona stima della testabilità del circuito appena descritto, indicandogli le procedure e le operazioni di difficile collaudabilità, diminuendo così il time to market. A sintesi avvenuta, inoltre, consente di identificare le parti del codice che generano porte non testabili rendendo più facile l'eventuale riprogettazione del circuito.

4.2 Usare RAGE

Scopo primario di un ATPG che funzioni a livello *gate* è quello di generare delle sequenze di ingressi, i pattern, tali da attivare il maggior numero possibile di guasti descritti nella lista dei guasti. Quest'approccio non funziona per descrizioni ad alto livello, descrizioni per le quali non è possibile creare una lista dei guasti. Per generare dei pattern che rilevino la testabilità di un circuito bisogna procedere in modo differente. RAGE ottiene i pattern di test attraverso tre fasi.

Nella prima fase RAGE mira a generare pattern che eseguano il maggior numero di operazioni. Il pattern migliore, ovvero quello che attiva più istruzioni, sarà usato come modello per le fasi successive e sarà anche il primo dei vettori di test. In questa fase vengono attivate le istruzioni più "facili" ovvero quelle più facilmente raggiungibili e eseguibili.

La seconda fase è mirata all'attivazione delle istruzioni che non si sono riuscite ad attivare nella prima fase. Viene selezionato un "target", un bersaglio che è l'operazione da attivare. Il genetico opererà al fine di trovare un pattern che attivi quest'istruzione un numero adeguato di volte.

La terza fase provvede a scartare le istruzioni che sono state raggiunte un numero sufficiente di volte mentre si procedeva all'attivazione di un'altra istruzione. Quello che si cerca di fare, quindi, è di trovare dei pattern che attivino tutte le istruzioni della descrizione ad alto livello. Questi saranno i test pattern.

La prima fase consiste nel compilare il sorgente VHDL. Questa fase è sempre necessaria per qualunque uso di RAGE.

4.3 ATPG – Automatic Test Pattern Generator

4.3.1 Usare RAGE come ATPG

RAGE funziona su circuiti sincroni. Gli ingressi quali clock e reset devono essere necessariamente specificati utilizzando gli appositi flag. RAGE ha un notevole numero di parametri che l'utilizzatore può variare a piacimento. Questi parametri modificano il comportamento del programma e del genetico contenuto al suo interno. Questo influirà sulla qualità dei risultati ottenuti e sulla velocità con cui essi si ottengono.

Il valore certamente più importante è il valore della soglia. La soglia indica il numero di volte che un segnale deve essere toccato da un singolo vettore di test affinché questo segnale sia considerato testato dal pattern stesso. Il valore di soglia è quindi un numero intero che può variare da 1 a cinque. Il valore zero non ha ovviamente alcun senso in quanto non porterebbe alcun risultato mentre valori più elevati sono possibili ma sconsigliati.

L'aumentare il valore di soglia dovrebbe permettere di ottenere dei pattern più validi ovvero dei pattern che meglio testano il circuito in esame. Questo è vero solo col crescere delle dimensioni del circuito; si verifica comunque un fenomeno di saturazione, ovvero, raggiunto un certo valore di soglia, non si hanno ulteriori incrementi delle performance aumentandolo. Per circuiti di piccole dimensioni (b01, b02) un valore di soglia pari ad uno si è dimostrato sufficiente mentre per circuiti più grandi (b04, b11, b14) si è utilizzato un valore pari a tre. Non si sono mai avuti incrementi di prestazioni per valori di soglia superiori ma nulla vieta di aumentarla per circuiti differenti. Un valore superiore a cinque comporterebbe però dei tempi di calcolo che poterebbero risultare eccessivi.

Il secondo parametro come importanza (l'unico che è necessario specificare sempre) riguarda la lunghezza iniziale dei pattern. Un circuito che presenti al suo

interno vari cicli e varie iterazioni necessiterà di sequenze più lunghe di un circuito lineare. Il valore consigliato per la maggior parte dei casi è di dieci.

L'algoritmo genetico provvederà comunque ad accorciare o ad allungare il pattern a seconda delle necessità. Un modo per allungare in maniera consistente i pattern consiste nel *block crossover*. L'attivazione di questa opzione provvede ad allungare le sequenze in maniera proporzionale al valore specificato durante l'attivazione. Qualora il genetico non riuscisse a trovare un pattern valido, inizierebbe ad allungare la nuova generazione di soluzioni attenendo soluzioni N volte più lunghe della sequenza padre.

Un valore valido deve essere superiore ad uno, altrimenti le sequenze si accorcerebbero, ed inferiore a tre, per evitare la generazione di sequenze troppo lunghe. I valori consigliati variano comunque tra 1,2 e due.

Ulteriori parametri influenzano il numero di individui costituenti la popolazione ed il numero di figli che devono essere prodotti ad ogni generazione. Un numero ridotto di individui e di figli rallenterebbe il processo evolutivo mentre un numero troppo elevato aumenterebbe il tempo di calcolo. Valori consigliati sono tra 10 e 50. Il numero di figli comunque non dovrebbe mai essere superiore a quello della popolazione. Queste limitazioni sono fornite dal buon senso e non da limitazioni del programma che lascia comunque la massima libertà. Come impostare questi parametri ed i comandi da fornire per compilare ed eseguire l'esecuzione sono forniti in appendice.

4.3.2 Risultati ottenuti usando RAGE come ATPG

Per valutare la qualità delle sequenze di test generate usando RAGE su una descrizione VHDL ci si è avvalsi del *fault simulator* e dell'ATPG utilizzati negli esperimenti precedenti, ovverosia quelli presenti nell'ambiente SUNRISE.

Questa volta però la descrizione a livello RT è stata fornita sia al sintetizzatore sia a RAGE e, una volta ottenute dal primo la lista dei guasti a livello *gate* e dal secondo le sequenze di test, si è calcolata la *fault coverage*.

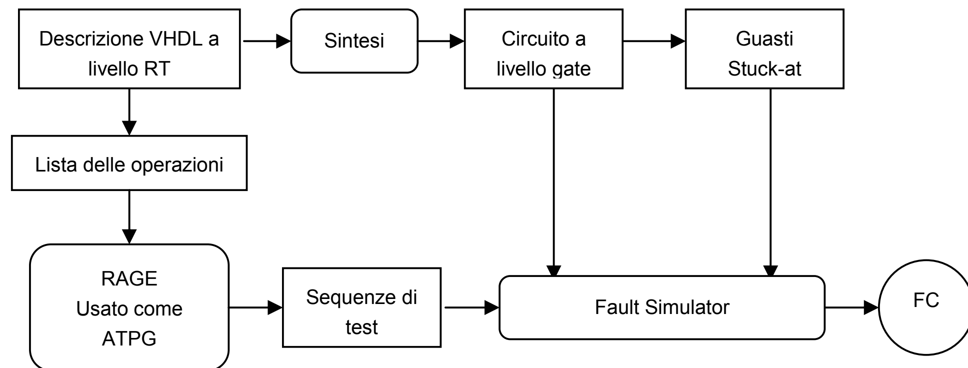

Figura 19: ***Flusso di valutazione di RAGE come ATPG.***

Per rendere confrontabili tra di loro i risultati ottenuti si è scelto di applicare RAGE a tutte le descrizioni con un valore di soglia pari a tre. Questa scelta non è la migliore in quanto i parametri del genetico dovrebbero essere scelti con cura in base alle caratteristiche del codice VHDL.

RAGE, come qualsiasi altro strumento per la progettazione automatica, prevede un'interazione col progettista, in particolare quest'ultimo, conoscendo le esigenze del progetto e le caratteristiche del circuito, decide se è meglio cambiare i parametri del genetico al fine di trovare una maggiore copertura o modificare la descrizione per ottenere una maggiore testabilità. Per facilitare questa interazione il programma fornisce in tempo reale la stima della copertura, *operation coverage*, e la percentuale di operazioni analizzate, così l'utente può decidere in ogni momento di fermare l'esecuzione perché è stata raggiunta la copertura voluta.

Nonostante il programma non funzioni nelle sue condizioni ottimali si può osservare, sia dai dati contenuti nella Tabella 6 sia dai grafici riportati in Figura 20, come le sequenze generate da RAGE ottengano una buona copertura dei guasti a livello *gate*.

	b01	*b02*	*b03*	*b04*	*b06*	*b09*	*b10*	*b11*
RAGE	100%	99%	73%	85%	94%	87%	91%	94%
ATPG	100%	99%	70%	90%	95%	90%	93%	74%
Differenze	0%	0%	3%	5%	1%	3%	2%	20%

Tabella 6: ***Fault coverage ottenuta con RAGE e con un ATPG a livello gate.***

Le differenze con un ATPG a livello gate variano intorno ai cinque punti percentuale e sono tutte ampiamente sopra l'80% di copertura eccezion fatta per l'esempio b3 in cui però la copertura di RAGE è migliore.

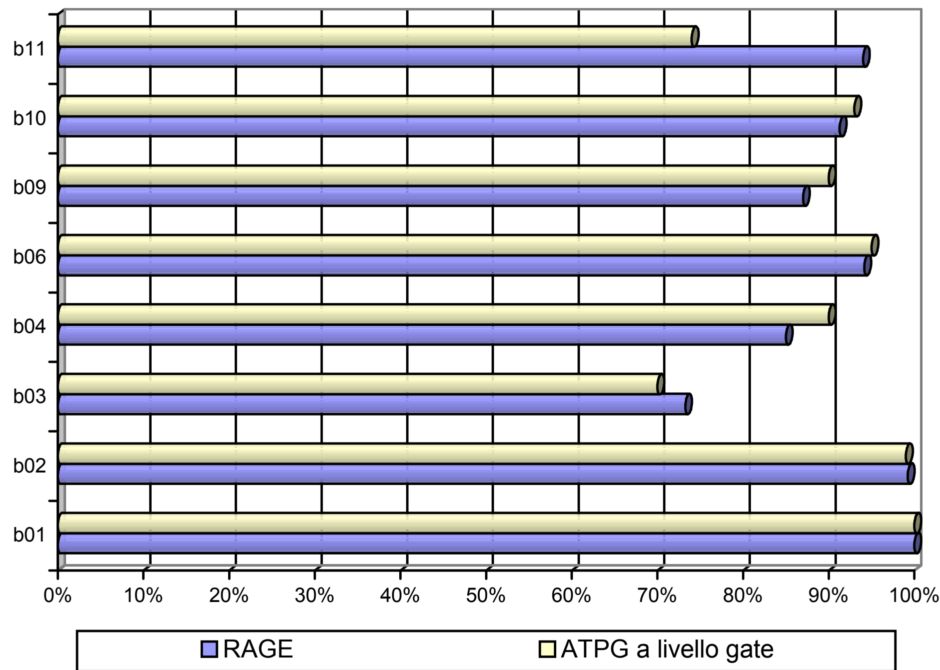

Figura 20: ***Fault coverage ottenuta con RAGE e con un ATPG a livello gate.***

Vale la pena soffermarci sui due esempi critici, b04 e b11, che hanno una differenza notevole fra il numero delle operazioni e quello dei guasti. In entrambi i casi si riesce ad ottenere una buona copertura nonostante la soglia non adeguata, ed in particolare il secondo circuito viene coperto con una percentuale il 20% maggire di quella ottenuta con l'ATPG a livello *gate.*

Per avere una visione più completa dei risultati è utile analizzare anche il tempo di CPU utilizzato dai due programmi per ottenere le sequenze di test. Il tempo impiegato da RAGE dipende da vari fattori, in particolare dal numero di operazioni da attivare, dalla complessità della descrizione VHDL del circuito e dal fatto che il programma si basa su un simulatore non commerciale e quindi meno ottimizzato.

	b01	*b02*	*b03*	*b04*	*b06*	*b09*	*b10*	*b11*
RAGE	26 s	12 s	58 s	37 s	39 s	1099 s	353 s	6826 s
ATPG	2,6 s	0,6 s	1280 s	783 s	3,5 s	1040 s	205 s	4074 s

Tabella 7: ***Tempistiche di RAGE e di un ATPG a livello gate.***

Come si può notare, dai dati contenuti in Tabella 7 e dai grafici di Figura 21, il tempo di CPU dipende molto dalla copertura raggiunta. Si noti come nell'esempio b11 il 20% in più ottenuto da RAGE costi in tempo quasi tremila secondi.

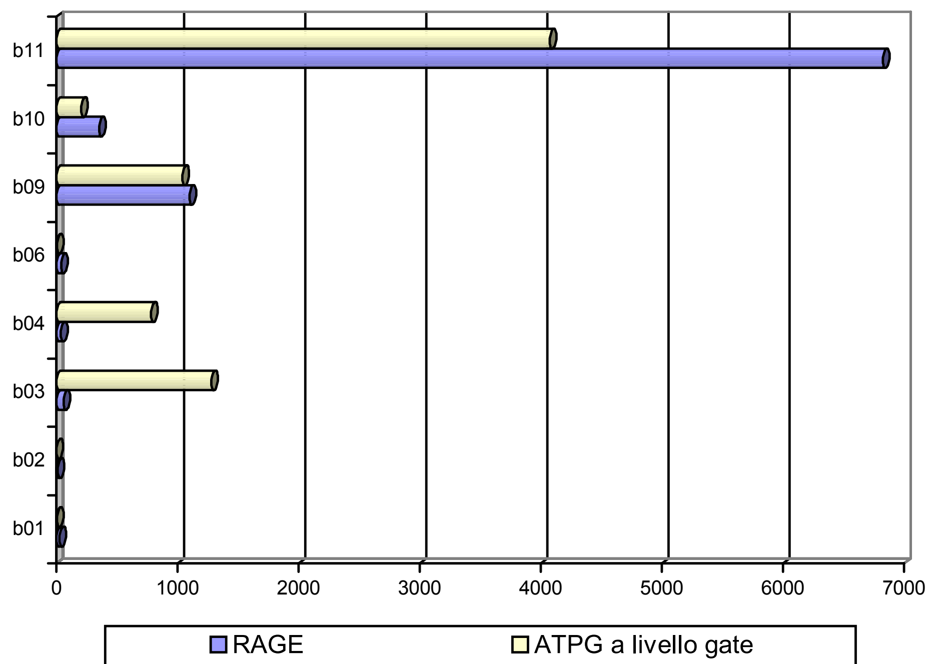

Figura 21: ***Tempistiche di RAGE e di un ATPG a livello gate.***

Come detto in precedenza è l'utente a scegliere se la copertura raggiunta è sufficiente ai propri scopi oppure è necessario alterare i valori del genetico, ed in particolare la soglia, per raggiungere valori maggiori. Si deve comunque tenere presente che spesso pochi punti percentuali in più nella *fault coverage* finale costano molto in quantità di tempo.

Come si può osservare dai grafici la crescita della *fault coverage* è notevole quando la soglia passa dal valore uno al tre mentre dal tre in poi i punti

percentuale guadagnati per ogni unità di soglia tende a diminuire. Dal grafico di Figura 23 si nota che di contro i tempi di CPU crescono notevolmente con l'aumento del valore di soglia.

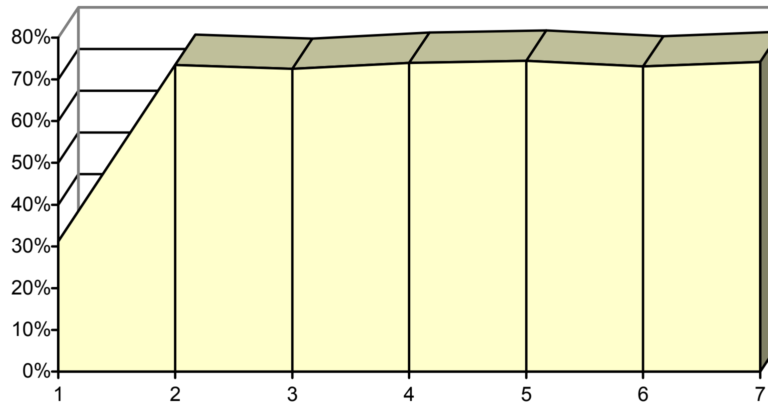

Figura 22: ***Variazione della fault coverage in base al valore di soglia.***

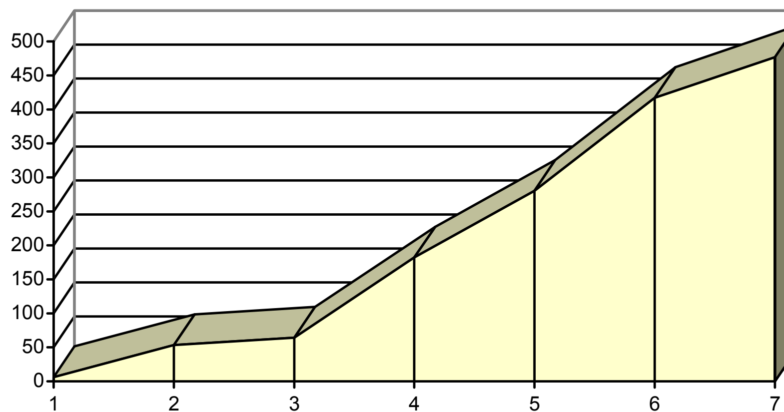

Figura 23: ***Variazione del tempo di CPU in base al valore di soglia.***

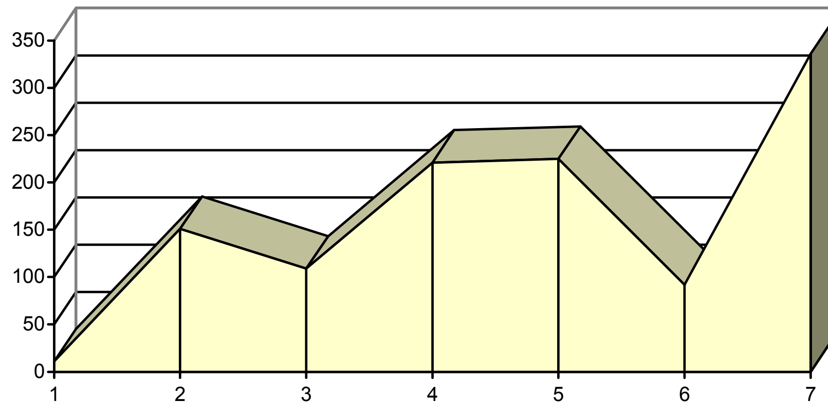

Figura 24: ***Variazione della lunghezza dei vettori di test in base al valore di soglia.***

4.4 Verifica di equivalenza

Come è stato illustrato nei precedenti capitoli, una descrizione in VHDL ad alto livello è sintetizzabile in un circuito fisico grazie a strumenti software reperibili sul mercato. È utile ricordare che questo processo è automatico e che agisce in funzione di come il codice è stato scritto.

Due differenti descrizioni producono due differenti layout anche se il comportamento descritto è il medesimo. Poiché la descrizione avviene ad alto livello, esistono differenti modi per rappresentare lo stesso circuito e quindi, a seconda di come viene scritto il codice, si avranno differenti layout incisi su silicio. In un certo senso è come se lo stile di programmazione influenzasse il risultato finale.

Ciò è vero anche qualora si scriva del software e non dell'hardware; creato un programma infatti si procede alla sua "ottimizzazione" cioè si cerca di migliorarlo in quei suoi aspetti funzionali ritenuti non completamente soddisfacenti. Per ciò che concerne l'hardware, le migliorie applicabili riguardano soprattutto la velocità di funzionamento e l'occupazione di area su silicio del circuito, circuito che deve mantenere sempre le medesime specifiche comportamentali. Un programmatore esperto sfrutterà la sua conoscenza del VHDL e del tool di sintesi per ottimizzare dei circuiti già esistenti e rendere quindi il proprio prodotto più competitivo e valido.

Esiste un progetto della Comunità Europea che si occupa di creare descrizioni che siano simulabili più velocemente. A questo progetto partecipa anche il Politecnico di Torino. Il compito che riguarda il Politecnico consiste nel verificare l'equivalenza comportamentale di due circuiti aventi diverse descrizioni in VHDL. Questo compito non è per nulla banale in quanto circuiti complessi possono presentare differenze comportamentali minime avvertibili solo in casi particolari. Poiché il compito di chi produce queste descrizioni è quello di crearne una che sia una versione migliorata della precedente ma con lo stesso comportamento, è ovvio

che non si avranno differenze macroscopiche di risultati, altrimenti lo stesso programmatore si accorgerebbe dell'errore, rimediandovi istantaneamente. Possono però sorgere differenze comportamentali in particolari situazioni, generalmente quelle che si verificano con meno frequenza, che possono sfuggire anche ad un occhio attento. Differenze di ritardo e piccole differenze in fase di controllo sono gli errori più frequenti ed anche i più difficili da individuare in quanto, generalmente, legati ad un pattern di attivazione unico e magari piuttosto complesso.

La possibilità di verificare l'uguaglianza delle due descrizioni in maniera esaustiva è ovviamente priva di senso in quanto richiederebbe una quantità sproporzionata di tempo così come per il collaudo di un circuito. Ma l'idea di utilizzare le stesse metodologie normalmente usate per il testing appare non priva di fondamento. Lo scopo è differente e quindi anche l'approccio al problema sarà differente ma le tecniche di base rimangono sostanzialmente invariate. La versatilità di RAGE e del suo algoritmo permette infatti di risolvere questo problema con una buona dose di sicurezza ed affidabilità in poco tempo. Analizziamo in dettaglio come si può usare RAGE per assolvere a questo scopo.

Come usare RAGE per la verifica di equivalenze

Poiché le due implementazioni sotto esame devono avere lo stesso comportamento, è ovvio che abbiano gli stessi ingressi e le stesse uscite. Non avrebbe senso un circuito che, pur elaborando gli stessi dati nello stesso modo, li acquisisca e li distribuisca in modi differenti dal circuito originale. Se così fosse esisterebbero già delle differenze tali da pregiudicarne la possibilità di interfacciamento con strutture già esistenti. Non si avrebbe più una ottimizzazione ma una completa reingegnerizzazione. Avendo gli stessi ingressi e le stesse uscite, invece, è possibile sostituire in un circuito già esistente la nuova versione migliorata del blocco in maniera del tutto indolore. Per verificare l'identità comportamentale, si possono quindi applicare una serie di pattern alle due strutture e confrontarne i risultati. Confrontare i risultati significa prendere le

uscite e verificarne l'uguaglianza bit a bit. Il circuito test da fornire a RAGE infatti deve essere modificato proprio seguendo questo criterio; si prendono le due descrizioni in VHDL, si uniscono i loro ingressi e si confrontano le uscite con una serie di confronti segnale per segnale, restituendo come unica, vera uscita del circuito un bit che indichi se sono state rilevate differenze o meno.

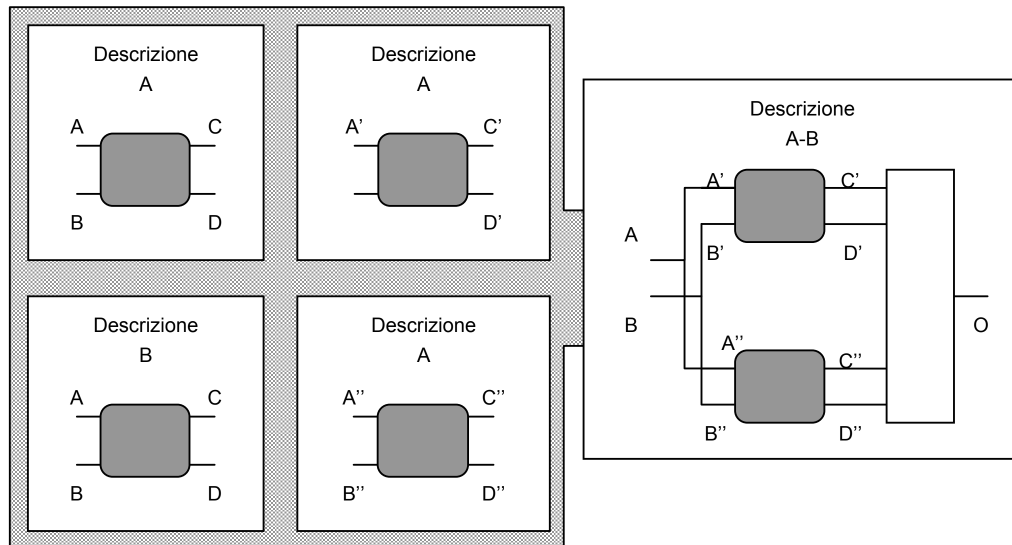

Figura 25: ***Creazione di una descrizione VHDL per la verifica di equivalenza.***

A questo punto è possibile procedere in due modi: cercare di attivare il bit d'uscita oppure attivare i segnali interni finché non si arrivi a rilevare una differenza. Qualora queste operazioni fallissero, i due circuiti verrebbero considerati identici dal punto di vista comportamentale. Analizziamo i pregi ed i difetti dei due differenti approcci: l’attivazione del bit rilevatore di differenze e l’attivazione di tutti i segnali interni.

La strada concettualmente più logica è quella che ha come scopo l'attivazione del bit rilevatore di differenze. L'algoritmo genetico cerca una popolazione di soluzioni a questo problema. Quest'approccio si rivela il più veloce ma anche il meno affidabile. Infatti ponendo immediatamente come target il bit d'uscita, la matrice delle adiacenze non ha tempo di svilupparsi completamente. Il genetico, quindi, non ha una funzione di fitness valida se non dopo un elevato numero di

tentativi durante i quali potrebbe anche ritenere che differenze non esistano e quindi fornire come risultato l'uguaglianza dei circuiti, risultato che potrebbe essere sbagliato. Questo in linea puramente teorica in quanto, con un adeguato settaggio dei parametri, tutti gli esempi pratici reali sono stati brillantemente risolti. La velocità con cui si arriva ad un risultato è notevole. Con il circuito fornitoci dai partner europei sono bastati pochi secondi di simulazione per ottenere un pattern che rilevasse delle differenze. Per ottenere maggiori ragguagli sulle tempistiche e sui risultati, si rimanda al paragrafo preposto.

Poiché con l'approccio visto precedentemente possono sorgere problemi, almeno in via teorica, si è pensato ad un metodo meno ortodosso e logico ma più sicuro per arrivare a dei risultati che potessero avere, in via teorica, una maggiore affidabilità. Il metodo in questione prevede l'attivazione di tutti i segnali interni dei due circuiti sotto esame. In questo modo si cerca di attivare ogni singola istruzione del codice e si verifica se il pattern che attiva l'istruzione in un circuito produce lo stesso valore in uscita nei due circuiti. In questo modo si cerca di produrre un pattern ben preciso che generi delle differenze comportamentali, istruzione per istruzione. In questo modo si saprebbe anche qual'è l'istruzione incriminata, cioè qual'è il punto esatto del circuito che genera la differenza. Quest'approccio è più dispendioso in termini di tempo ma è più accurato e fornisce risultati più precisi.

La simulazione continuerebbe sino all'esaurimento dei segnali o finché non si riesca ad individuare una differenza. Negli esempi pratici non si è mai sentito il bisogno di un approccio di questo tipo ma ci è parso comunque utile fornire la possibilità di ricorrervi qualora se ne presenti la necessità.

4.4.1 Risultati ottenuti usando RAGE per la verifica di equivalenze

Per sfruttare RAGE nella verifica delle equivalenze occorre creare una descrizione VHDL che contenga i due codici insieme ad alcune istruzioni che facciano il confronto tra le uscite dei circuiti. Così facendo basta cercare di

attivare l'operazione che fornisce in uscita il segnale che rivela una differenza tra le descrizioni e si ottiene la sequenza di test critica per l'equivalenza.

Solitamente basta coprire una singola istruzione scartando le altre, in questo modo l'esecuzione risulta molto veloce. Per circuiti di grande complessità però è utili usare RAGE come fosse un ATPG, cercando di attivare tutte le operazioni, in questo modo il genetico ha una probabilità maggiore di rivelare l'eventuale differenza. Si ricorda, infatti, che all'interno dell'algoritmo è presente una procedura di apprendimento e quindi spesso è sufficiente la presimulazione a fornire al genetico i dati sufficienti per funzionare al meglio, ma più l'algoritmo lavora più la matrice di adiacenza si perfeziona e meno tempo impiegherà a trovare le sequenze giuste.

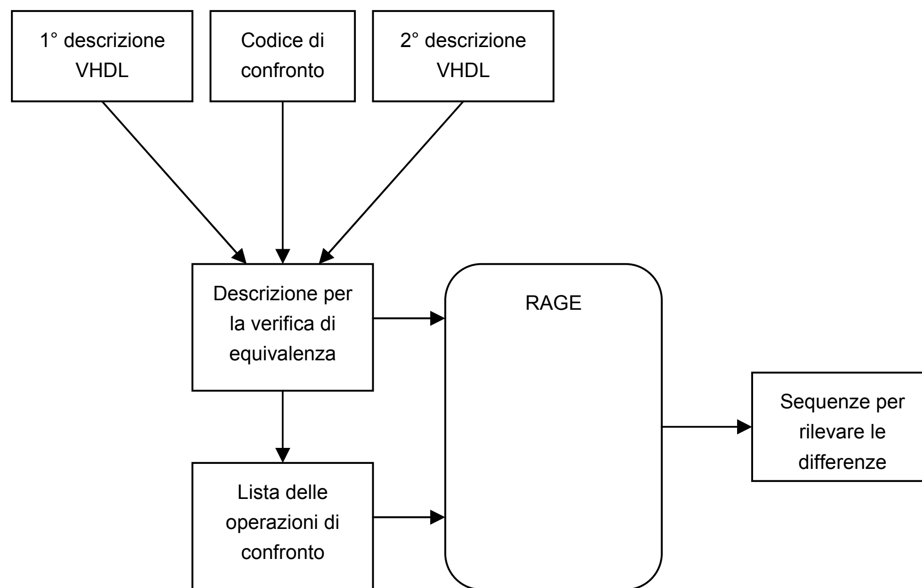

Figura 26: ***Flusso di valutazione di RAGE usato per la verifica delle equivalenze.***

Per collaudare il buon funzionamento di RAGE come strumento per la verifica di equivalenza si sono modificate leggermente le descrizioni di test introdurre piccole differenze. Le variazioni sono state pensate per rendere problematica la loro individuazione, procedure difficili da raggiungere oppure variazioni che hanno effetto dopo molte iterazioni.

Nonostante tutto i nostri tentativi di mettere in crisi il programma sono stati infruttuosi, in quanto RAGE è riuscito sempre a trovare una sequenza per cui il comportamento delle due descrizioni non fosse identico.

Per i suoi pregi RAGE si sta occupando di svolgere queste verifiche anche per il progetto europeo FOST, in particolare viene utilizzato per vedere se descrizioni che sono simulabili più velocemente di altre svolgano veramente le stesse operazioni.

Conclusioni

Questa tesi ha presentato un nuovo tipo di approccio alla verifica dell'osservabilità dei guasti riscontrabili in circuiti digitali descritti ad alto livello ed alla generazione di vettori di collaudo che fossero in grado di garantire un collaudo accurato del circuito medesimo. La ricerca dei vettori di collaudo è stata effettuata eseguendo le istruzioni del codice VHDL ed in particolare scrivendo e leggendo i registri. Particolare attenzione è stata posta nell'attivare quelle istruzioni di difficile esecuzione cercando così di eccitare anche i difetti più nascosti e fornendo al contempo al progettista una indicazione valida sulla reale raggiungibilità delle istruzioni del codice da lui scritto. Questo nuovo approccio al collaudo ha inoltre dimostrato un legame tra la controllabilità delle istruzioni VHDL e l'osservabilità dei guasti rilevabili a livello più basso.

I vantaggi di questo nuovo approccio risiedono nella possibilità di ottenere informazioni importanti già nelle prime fasi di progetto facilitando così eventuali modifiche; il raggiungimento di un elevato grado di testabilità è quindi possibile fin dai primi momenti e non è più necessario attendere l'implementazione del circuito per ottenere misure accurate. Questo si traduce in un risparmio considerevole di sforzi, di tempo e conseguentemente di denaro. In alcuni casi questo tipo di verifica risulta l'unica possibile; qualora si acquistino componenti da terze parti e non sia fornita una descrizione del componente stesso a livello gate, è possibile comunque verificare la testabilità del circuito ricostruendone una descrizione comportamentale.

La versatilità dell'algoritmo sviluppato ne permette l'uso anche in campi d'interesse differenti da quello del collaudo. Il suo utilizzo al fine di rilevare differenze comportamentali tra descrizioni differenti dello stesso circuito ha prodotto risultati lusinghieri. Questa verifica è parte integrante di un progetto che mira al miglioramento delle prestazioni dei circuiti ottenibili mediante sintesi automatica.

Lo sviluppo di un apposito tool ha permesso di effettuare prove e di ottenere dei risultati sperimentali. Questi risultati hanno dimostrato la bontà di questo algoritmo che, unito alle capacità di ricerca proprie degli algoritmi genetici, ha permesso al prototipo da noi sviluppato, applicato ad un sottoinsieme significativo di descrizioni VHDL, il raggiungimento di prestazioni pari ed in alcuni casi superiori a quelle degli strumenti commerciali.

Bibliografia

[ABFr90] M. Abramovici, M. A. Breuer, A. D. Friedman: *Digital systems testing and testable design*, Computer Science Press, New York, NY (USA), 1990

[Arms93] J.R. Armstrong: *Hierarchical test generation: where we are, and where we should be going*, Euro-DAC'93: European Design Automation Conference with Euro-VHDL, Hamburg (Germany), September 1993, pp. 434-439

[CaBr92] J. Calhoun, F. Brglez: *A Framework and Method for Hierarchical Test Generation*, IEEE Transactions on Computer-Aided Design, Vol. 11, N. 1, January 1992, pp. 45-67

[ChKr96] K.-T. Cheng, A.S. Krishnakumar: *Automatic Generation of Functional Vectors Using the Extended Finits State Machine Model*, ACM Trans. on Design Automation of Electronic Systems, January 1996, Vol. 1, No. 1, pp. 57-79

[CPRS96] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, *GATTO: a Genetic Algorithm for Automatic Test Pattern Generation for Large Synchronous Sequential Circuits*, IEEE Transactions on Computer-Aided Design, August 1996, Vol. 15, No. 8, pp. 991-1000

[Gold89] D.E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley, 1989

[LEDA95] LVS System User's Manual, LEDA Languages for Design Automation, Meylan (F), April 1995

[LePa93] J. Lee, J. Patel: *Testability Analysis Based on Structural and Behavioral Information*, IEEE VLSI Test Symposium, April 1993, pp. 139-145

[Mart93] A.R. Martello, A VHDL Design Environment, Technical Report, University of Pittsburgh, Department of Electrical Engineering, June 1993

[Sunr95] Sunrise Reference Manual, *Sunrise Test Systems*, 1995

[Syno94] VHDL Compiler Reference Manual, Synopsys Inc.

Indice analitico

A

Algoritmi Genetici....................................32

Algoritmo 38

ARMAGEDDON 47

ATPG.. 63; 64

B

behavior ..... *Vedi* domini di rappresentazione

benchmarks..58

block crossover..64

BRAHMA..41

C

circuiti di prova ....*Vedi* benchmarks

collaudo 9; 22

costo...17

crossover

block...46

orizzontale 43

verticale...43

D

device..................... *Vedi* livelli di astrazione

domini di rappresentazione 13

F

fault coverage 56

fault dropping 41

fault list..56

fitness.. 34; 50

roulette 35

torneo 35

G

garbage collector 53

Genetico................. *Vedi* Algoritmi Genetici

Guasti

stuck at ....*Vedi* stuck at

guasto...22

I

Implementazione 48

L

livelli di astrazione 13

logic .... *Vedi* livelli di astrazione

M

matrice di adiacenze 50

modelli di guasto 25

mutazione 44

aggiuntiva 44

sottrattiva 45

O

operation coverage 59; 65

P

parametri di costo ....*Vedi* costo

physical...... *Vedi* domini di rappresentazione

Presimulazione 51

R

reborn rate...47

S

SHIVA 46

simulazione...29

Simulazione 36; 52

Sintesi 17

soglia.. 40; 50

structure ..... *Vedi* domini di rappresentazione

stuck at 25; 38

system .... *Vedi* livelli di astrazione

T

tecnologia..17

tempo di CPU 66

testabilità... 58; 73

Testabilità 22

time to market...23

torneo 43

V

VHDL 13

vincoli 18

VISNU 42

Indice delle figure

Figura 1: Livelli di astrazione e domini di rappresentazione....12
Figura 2: Processo di sintesi. ...17
Figura 3: Classificazione delle possibili sintesi....19
Figura 4: Classificazione degli strumenti di sintesi....20
Figura 5: Il time to market....22
Figura 6: Impatto del time to market sui profitti...23
Figura 7: Codifica di un individuo o sequenza. ...42
Figura 8: Schema di funzionamento del crossover verticale....43
Figura 9: Schema di funzionamento del crossover orizzontale. ...44
Figura 10: Operazioni di mutazione. ...45
Figura 11: Schema di funzionamento del block crossover con valore 2....46
Figura 12: Creazione del simulatore...48
Figura 13: Flusso di valutazione....53
Figura 14: Confronto fra i risultati ottenuti con le due diverse fitness....54
Figura 15: Esempio di sequenza di test....57
Figura 16: Flusso di valutazione di RAGE nell'analisi di testabilità....58
Figura 17: Confronto fra l'operation coverage fornita da RAGE e la fault coverage. ...59
Figura 18: Rapporto tra linee di descrizione e porte sinstetizzate....60
Figura 19: Flusso di valutazione di RAGE come ATPG....65
Figura 20: Fault coverage ottenuta con RAGE e con un ATPG a livello gate. ...66
Figura 21: Tempistiche di RAGE e di un ATPG a livello gate....67
Figura 22: Variazione della fault coverage in base al valore di soglia....68
Figura 23: Variazione del tempo di CPU in base al valore di soglia. ...68
Figura 24: Variazione del tempo di CPU in base al valore di soglia. ...69
Figura 25: Creazione di una descrizione VHDL per la verifica di equivalenza....72
Figura 26: Flusso di valutazione di RAGE usato per la verifica delle equivalenze. ...74

Indice delle tabelle

Tabella 1: Esempio di lista delle operazioni. 49
Tabella 2: Esempio di matrice di adiacenze per l'esempio in Tab. 1. 50
Tabella 3: Circuiti di prova descritti a livello RT. 58
Tabella 4: Confronto fra l'operation coverage fornita da RAGE e la fault coverage. 59
Tabella 5: Rapporto tra linee di descrizione e porte sinstetizzate. 60
Tabella 6: Fault coverage ottenuta con RAGE e con un ATPG a livello gate. 65
Tabella 7: Tempistiche di RAGE e di un ATPG a livello gate. 67