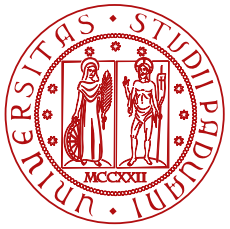

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Laurea magistrale in Ingegneria Elettronica

Sviluppo di una libreria UVM per la verifica digitale di un PMIC

Laureando

Giorgio Olivero

ID studente 1128282

Relatore

Prof. Daniele Vogrig

Universita' di Padova

Tutor aziendale

Ing. Virginia Cananzi

Infineon Italia

Anno accademico
2021/2022

Sommario

I circuiti elettronici sono sempre più presenti anche dove prima non lo erano, basti pensare al settore Automotive e come al giorno d'oggi ogni auto ha una vasta componente elettronica che vent'anni fa non era presente. Proprio in questo settore, e non solo, è fondamentale che i cuircuiti che vengono utilizzati siano ampiamente testati per evitare incidenti e tutelare l'azienda produttrice e soprattutto il consumatore. Lo scopo dei test è proprio quello di trovare dei problemi tenendo conto che più avanti si va con il progetto, maggiore sarà il costo per risolvere tali problemi.Costruire un ambiente di test infatti è un processo che richiede un tempo molto lungo, basti pensare che in media un progetto dura un paio di anni (tra design e verifica). In questo periodo viene anche allocata una porzione necessaria per imbastire l'ambiente per il test e generalmente richiede qualche mese di lavoro. Questo progetto nasce proprio dalla necessità di essere efficienti nelle tempistiche perchè il tempo che viene "perso" inizialmente per creare un ambiente potrebbe essere utilizzato per testare più approfonditamente il circuito in esame. Lo scopo di questa tesi, partendo da un progetto reale, è proprio quello di sviluppare una libreria base che consenta di creare un ambiente di test in facilità e velocità. In particolare la libreria permette di modellare le varie strutture all'interno del circuito(fornendogli funzioni specifiche in base al modello), gestire gli errori che compaiono nei modelli, monitorare i vari segnali del DUT(Device Under Test), creare e gestire una macchina a stati finiti. Tutte queste funzioni sono ampiamente documentate e sono forniti degli esempi per spiegare come utilizzarle. La libreria in futuro può essere ampliata per fornire funzionalità presenti in nuovi lavori, così da implementare una politica volta al riutilizzo del codice invece di ripartire da zero ogniqualvolta ci sia un nuovo progetto.

Indice

1 Introduzione

In questo capitolo vengono presentati gli obbiettivi che ci si erano prefissati all'inizio della tesi e poi viene descritto quello che verrà trattato più approfonditamente nei capitoli successivi.
La tesi è stata svolta presso Infineon Technologies Italia, azienda italiana leader nel settore automotive, specificatamente nell'ambito di produzione di chip utilizzati all'interno dell'elettronica che va a comporre una macchina.

1.1 Obbiettivi

Il periodo di tesi si suddivide in due parti: la prima è dedicata ad uno studio approfondito del System $Verilog$[1] e della libreria UVM[2], che oggigiorno sono uno standard del settore. La seconda parte invece è rivolta allo studio di un progetto già terminato che ha l'obiettivo di creare una libreria di classi riutilizzabili in progetti futuri.
Il progetto di partenza è un ambiente di test di un PMIC nel quale sono presenti molte classi e dove vengono testate varie configurazioni per trovare degli errori che comportano sia ritardi di tempo molto lunghi che costi di modifica molto elevati. Basti pensare che se viene trovato un bug facendo la verifica prima di creare un prototipo si risparmia tempo e denaro.
Inizialmente è stata studiata la struttura del progetto per capire come i vari blocchi erano collegati e con l'aiuto del mio tutor aziendale, l'Ing.Virginia Cananzi, mi è stato spiegato un pò alla volta il loro funzionamento.
Ognuno di questi blocchi fondamentali è stato analizzato nello specifico ed è stata ricavata una classe generale, riutilizzabile anche in altri progetti, che è stata reintrodotta nel progetto di base, ovviamente adattandolo nel suo intero alle modifiche e testando che tutto funzionasse correttamente.
Questo processo è stato seguito per tutte le classi che sono spiegate approfonditamente

nei seguenti capitoli.
Inoltre alla fine è stato creato anche un ambiente di test in parallelo che utilizza tutte le classi che sono state inserite in questa libreria.

1.2 Infineon Technologies Italia

La tesi è stata svolta all'interno dell'azienda Infineon Thechnologies Italia, multinazionale nel settore automotive, nella sede di Padova.
All'interno dell'azienda sono presenti molti gruppi che lavorano generalmente su progetti diversi nel campo dell'elettronica e in particolare il mio tirocinio è stato svolto nel gruppo di verifica digitale guidato dall'Ing. Florindo Santoro.
Il gruppo è composto non solo da persone che lavorano nel campo della verifica ma anche dal lato del progetto e sintesi di circuiti digitali.
Nello specifico il gruppo di verifica è incaricato di testare la parte digitale e vengono forniti dei modelli per la parte analogica, il funzionamento di questi modelli viene controllato da altri gruppi presenti in azienda.
Per rendere efficiente il lavoro ogni lunedì viene effettuata una riunione in cui ogni membro del gruppo spiega quello che ha fatto durante la settimana passata, con eventuali problematiche riscontrate, e spiega quello su cui lavorerà nella settimana corrente.
Per il mio progetto ho lavorato sotto le direttive dell'Ing. Virginia Cananzi, Senior Digital Verification Engineer per Infineon dall'ottobre del 2016, la quale mi ha spiegato tutto quello che dovevo sapere per l'utilizzo del software aziendale ed è rimasta disponibile per qualsiasi problema riscontrassi.
Durante il corso del tirocinio sono state svolte delle review, con l'Ing. Cananzi, della libreria da me sviluppata e anche per discutere eventuali migliorie del codice.

1.3 Le classi della libreria

Adesso descriviamo un pò le varie classi e le loro funzionalità e come esempio prendiamo quello fornito nel capitolo 8 in cui viene costruito un ambiente di test parallelo.
Parte tutto dalla classe dei modelli, capitolo 4, nella quale è presente una gestione degli errori tramite delle funzioni fornite.
E' infatti possibile lanciare una gestione degli errori da questa classe ma anche scaturire dei cambi di stato lanciando un evento che viene "raccolto" dalla macchina a stati finiti, capitolo 5, che contiene tutti gli stati che possono essere raggiunti e ne gestisce anche la transizione.
La domanda che può sorgere spontanea è come scaturire questi eventi e la risposta risiede nel monitor, capitolo 6, che è connesso all'interfaccia e generalmente monitora certi segnali.

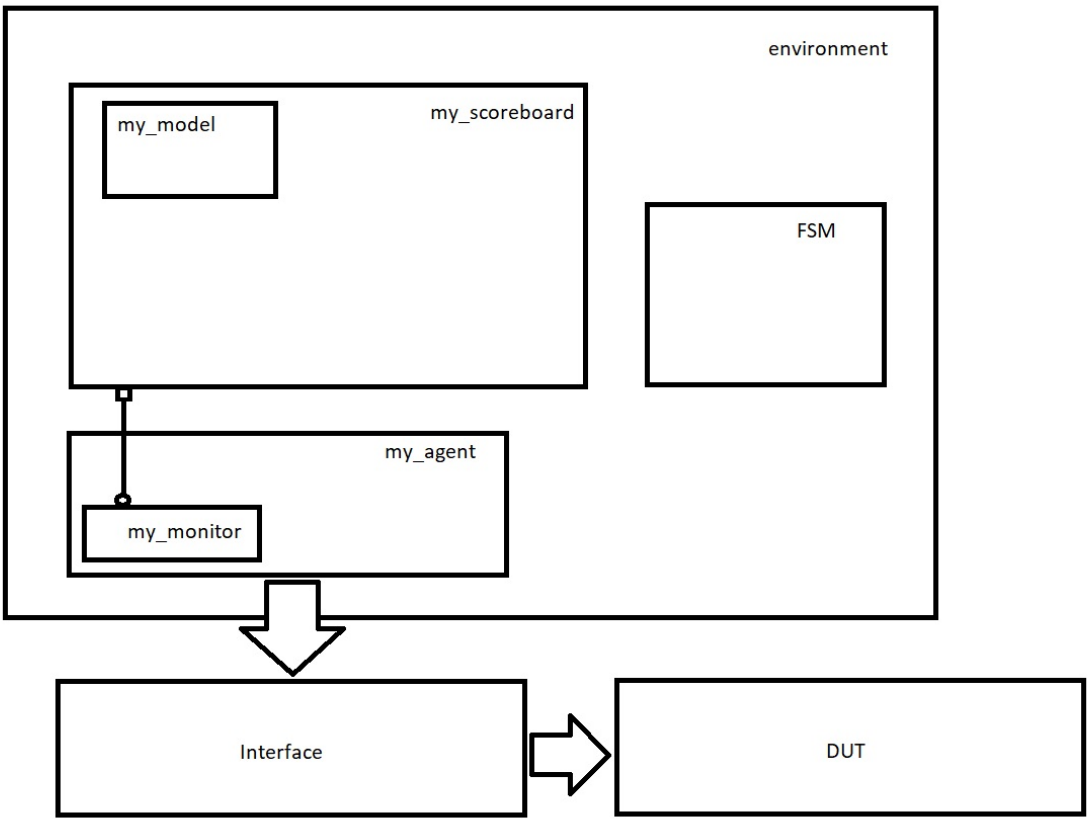

Figura 1.1: Ambiente parallelo

Quando questi segnali cambiano viene mandato un oggetto allo scoreboard, capitolo 7, il cui compito è quello di comunicare ai modelli cosa è necessario fare in base a quello che il monitor rileva.
Supponiamo che il monitor rilevi un cambio anomalo di un segnale, in questo caso viene segnalato allo scoreboard che nei suoi metodi di gestione deve avvisare il modello o i modelli.
Nello scoreboard infatti sono presenti tutti i modelli e si possono richiamare le funzioni che gestiscono gli errori,in questo caso si suppone che la chiamata della funzione del modello faccia scaturire un cambio di stato che verrà comunicato alla FSM.

2

Introduzione al System Verilog

2.1 Predecessori

Prima di parlare del System Verilog bisogna introdurre le motivazioni per le quali ci sia stata un'evoluzione verso questo linguaggio.
Innanzitutto prima di questo esistevano due linguaggi principali, il Verilog e il VHDL, ed entrambi fornivano sufficienti funzionalita' per progettare e sintetizzare circuiti digitali.
Il problema del VHDL risiede nel fatto che è possibile solamente creare test diretti: è necessario crearlo e ha degli input specifici, succesivamente vengono analizzati i segnali che producono tali ingressi per trovare errori. Non è possibile rendere gli ingressi variabili. Questo può essere un problema perchè se parliamo di un circuito molto semplice con magari due bit di input e due di output sono facilmente testabili tutte le possibili combinazioni.
Se ho un circuito complesso con 40 ingressi e più moduli indipendenti connessi fra loro risulta impossibile testare tale circuito con un approccio del genere e la stessa cosa vale anche per il Verilog.
Per questo motivo è stato creato un linguaggio di programmazione, il System Verilog, che assimila la parte di progetto e sintesi a quella di verifica.
Tutto questo è semplificato dall'introduzione della programmazione ad oggetti e dalla libreria UVM che permettono una creazione di un ambiente di test, con una introduzione di stimoli casuali che permette di sollecitare il circuito a diversi input senza il bisogno di creare nuovi test.
Ovviamente introducendo il concetto di casualità viene da sè che non è possibile, la maggior parte delle volte, testare il circuito per qualsiasi input, generalmente per le limitazioni temporali legate al progetto.
Per queste ragioni talvolta si ricorre a test diretti per testare casi particolari che non sono emersi dalla simulazione randomica.

2.2 Tipi di dati

Il System Verilog ha delle sostanziali differenze rispetto ad altri linguaggi, in particolare rispetto ai suoi predecessori, che gli permettono di essere più efficiente e versatile soprattutto per quanto riguarda la verifica.

2.2.1 Logic, Wire e bit: le differenze

I tipi di dato *logic* e *wire* possono assumere tutti e quattro i valori che sono presenti nella tabella qui sotto. La differenza sostanziale è che il *wire* viene utilizzato per connettere ingresso e uscita di due diversi moduli.
Il tipo di dato *bit* invece rispecchia i valori che può assumere un qualsiasi segnale digitale, che non può essere altro che alto o basso (massa e alimentazione).

Valore	Spiegazione
0	Valore logico basso(Massa)
1	Valore logico alto(Alimentazione)
x	Valore non conosciuto
z	Alta impedenza(non connesso)

2.2.2 Arrays

In System Verilog esistono due tipi di array: *packed* e *unpacked*. Qui sotto è mostrato come dichiarare i due tipi.
Possono anche essere dichiarati dinamici, che vuol dire che quando vengono creati bisogna passare loro il numero di elementi che voglio. Inoltre una "pecca" di questo linguaggio di programmazione è che se dichiaro un array, i limiti devono essere delle costanti e non possono essere definiti in fase di esecuzione magari in base a qualche condizione.

```
1 bit [1:0][3:0] packed_array;
2 bit unpacked_array [7:0];
3 packed_array[0]=4'b1111 // metto un valore nella "prima cella"
4 unpacked_array[0]=1'b1; // scrivo il valore logico 1 nella prima cella
5 int time_to_wait [string];// questo crea un array di tipo unpacked che andrà inizializzato ed è possibile accedere ai diversi
      valori tramite una stringa
6 int array_1 [];//array dinamico
7 array_1=new(5);
```

La differenza sta nel fatto che un *unpacked* array viene messo in memoria in un unico indirizzo ed è possibile accedere ad una singola cella.
I *packed* array invece hanno un indirizzo per ogni sottoporzione. Per prendere un esempio reale, vediamo quello sopra: da quella dichiarazione viene creato un array diviso in due porzioni da 4 bit ciascuna ed è possibile accedere direttamente ad un blocco di

bit.
C'è anche un'altra differenza fra i due: gli *unpacked* array possono essere creati per qualsiasi tipo di dato mentre gli altri no, per esempio con le stringhe non è possibile.

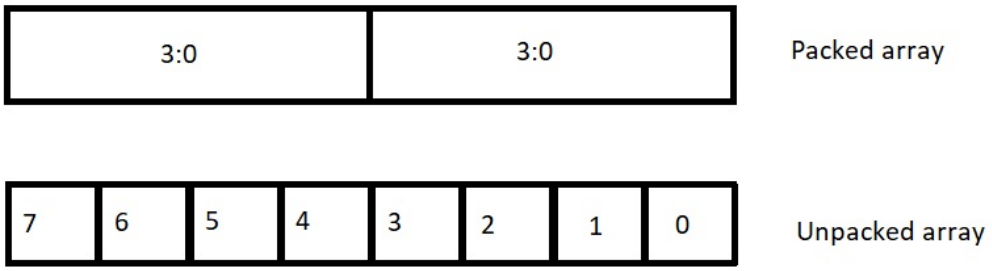

Figura 2.1: Rappresentazione grafica

E' possibile scorrere all'interno di un array in vari modi, qui sotto ne vengono elencati alcuni. Le librerie del System Verilog forniscono inoltre alcuni metodi utili, e già pronti, come per esempio la ricerca di minimo e massimo.

```
bit unpacked_array [7:0];

for(int i=0:i<=unpacked_array.size();i++) begin
    unpacked_array[i]=1;
end

//questa soluzione è la più semplice e l'unica che si può utilizzare nel caso in cui non si abbia a che fare con interi ma con
      magari stringhe senza conoscere a priori i valori degli indici
foreach(unpacked_array[i]) begin
    unpacked_array[i]=1;
end

int i=0;
while(i<=unpacked_array.size()) begin
    unpacked_array[i]=1;
    i++;
end
```

2.2.3 Code

Le code sono un'alternativa agli array, permettendo di fare delle operazioni particolari. Sono una struttura dati denominato FIFO(First In First Out) che viene prevalentemente usata quando parliamo di processi.

Come è possibile vedere nella figura in generale vengono usate in questo modo, però la struttura permette anche di togliere e/o mettere un elemento sia dalla fine che dall'inizio della coda permettendo così all'utente di decidere come usarla.

```
int queue_1 [$];
queue.push_back(10);//[10]
queue.push_front(1);//[10,1]
queue.pop_back();//[1]
queue.push_front(23);//[23,1]
queue.pop_front();//[23]
```

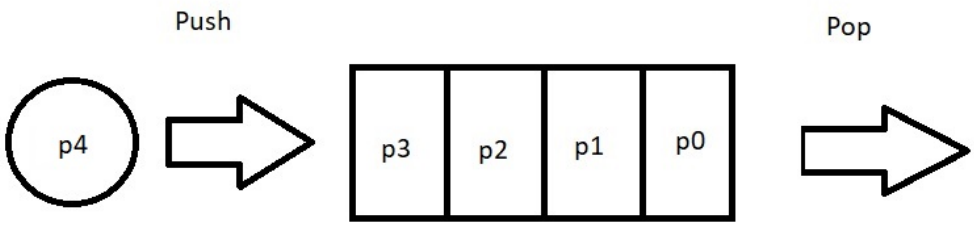

Figura 2.2: Operazioni sulle code

E' anche possibile visitare ogni elemento di una coda utilizzando un *for each* come visto per gli array.

2.2.4 Enumerazioni

Il tipo *enum* è un tipo particolare di dato che permette di associare un "titolo" ad un numero. Questo tipo di dato viene utilizzato nei test per svariati motivi, per esempio per dare dei nomi ai vari stati della macchina a stati finiti.

```
enum {INIT,FAILSAFE,POWERDOWN} enum_1;//in questo caso viene creato un'enumerazione a cui vengono assegnati INIT=0,FAILSAFE=1,
    POWERDOWN=2

enum {INIT=7,FAILSAFE=2,POWERDOWN=0} enum_2;//è possibile anche decidere che valore assegnare ai vari enum
```

Una volta creati, invece che dover conoscere a cosa si riferisce ogni singolo numero, è possibile farne riferimento utilizzando il valore (*INIT*,*FAILSAFE*,*POWERDOWN*).
Una particolarità di questo tipo di dato è che in un ambiente di lavoro non è possibile creare due enumerazioni che contengono lo stesso valore (non quello numerico ma il nome di riferimento) perchè altrimenti il compilatore non è in grado di distinguere a quale si riferisce.
Una funzionalità molto utile è questa: pensiamo di essere nel caso in cui ho una serie di dati che mi arrivano da una porta, può essere considerata come un canale in cui vengono spediti dei pacchetti. Quando questi pacchetti vengono ricevuti al loro interno c'è un enumerazione e voglio per esempio aspettare un certo tempo diverso per ogni valore, con gli *enum* è possibile creare un array che contiene questi tempi differenziati per ogni tipo.

```
enum {BOOST,BUCK,WWD} enum_3;
int time_array [];//array dinamico
time_array=new(enum_3.num());//crea un array con 3 elementi
time_array[BOOST]=3;
time_array[BUCK]=2;
time_array[WWD]=5;
```

2.2.5 Creazione di nuovi tipi

Un'altra cosa che si può fare in System Verilog è quella di definire un tipo. Per esempio se ho una definizione ricorrente è possibile creare un tipo e invece di rifarla ogni volta posso assegnare questo.

```
typedef enum bit [1:0] {BOOST,BUCK,WWD} model_t; // definisco il tipo
model_t specific_type;//creo una variabile del tipo model_t
```

All'interno dello stesso file posso creare variabili di questo tipo, se voglio utilizzarlo al di fuori devo generalmente creare un package e importarlo in tutti i file in cui viene utilizzato. Viene mostrato nel paragrafo successivo come fare.
Una struttura molto utile è quella dello *struct*, che consente di creare degli oggetti che hanno più campi (per capirci una classe, ma senza metodi).

```
typedef struct {
    int id;
    int count;
} struct_t; // definisco il tipo
```

Questo è un esempio in cui contiene un *id* (generalmente univoco) e un *count* (che verrà incrementato se succede un evento).
Come per gli array, la struttura può essere *packed* o *unpacked* e come abbiamo visto prima, solo certi tipi di dato possono andare in un *packed* array.

2.2.6 Packages

Molto spesso è necessario utilizzare delle variabili o delle classi in varie parti del progetto. Un modo per farlo è tramite la creazione dei *package*.
Una volta creati possono essere importati ovunque siano necessari, bisogna sempre avere attenzione che non ci siano altre variabili e/o classi con lo stesso nome che ovviamente impediranno la compilazione di tale progetto.
Supponiamo di volere importare un file di nome *"common_variables.svh"*, che contiene le variabili comuni, e spieghiamo come si crea un pacchetto che lo contenga.
Questo pacchetto va messo in un file a sè stante, per semplicità si suppone che il file del package (per esempio *"common_variables_pkg.sv"*, notare la diversa estensione perchè solitamente i package devono avere estensione .sv per far si che il compilatore li consideri tali e li renda visibili e importabili nel progetto) sia nella stessa cartella, altrimenti basta modificare il percorso.

```
1 package common_variable_pkg;
2 `include "common_variables.svh"
3 endpackage
```

Adesso per utilizzare il package in un qualsiasi file basta importarlo.

```
1 import common_variable_pkg::*;
```

I due punti seguiti dall'asterisco indicano al compilatore di importare tutte le classi, però si potrebbe importare anche solo qualche oggetto specifico (in questo modo rendo evidente cosa sto usando di quel package).
Da notare come viene importato il file nel package perchè potreste incorrere in un problema: verrebbe da dire che basta usare un *include* come sopra e metterlo dove si vuole, ma si farebbe un errore. Quello che fa la parola chiave *include* è una copia vera e propria del codice quindi se viene messo in più punti, effettivamente ho più copie della stessa variabile/classe e il compilatore darà errore.

2.3 Funzioni

Più o meno in tutti i progetti sono presenti delle funzioni, System Verilog ha qualche particolarità per come le gestisce e nei prossimi paragrafi faremo una carrellata di tutte le cose che si possono o non si possono fare.

2.3.1 Differenza fra task e function

La sostanziale differenza è che le *function* possono contenere solamente righe di codice che sono sequenziali, ovvero che vengono eseguite tutte senza un delay di qualsiasi tipo.
D'altra parte i *task* permettono di avere delay e utilizzare molte strutture che non sono permesse nelle funzioni.
Qui sotto faccio un esempio di un *task*. La parola chiave che si chiama *ref* fa in modo che venga passato l'indirizzo della variabile perchè altrimenti entrando nel forever se il segnale *wait_signal* cambia il task non se ne accorge. Si potrebbe fare la stessa cosa per la variabile *can_break*.

```
task automatic wait_and_see(ref logic wait_signal,input bit can_break);
    //questa struttura non ha fine e una volta lanciata va sempre a meno che non venga utilizzato un break; che la interrompe
    forever begin
        @(wait_signal)//quando questo segnale cambia posso andare avanti con l'istruzione successiva
        #50ps;//aspetto un tempo pari a 50 picosecondi
        if(can_break) break;
    end
endtask
```

2.3.2 Definizione automatic

Nella funzione del paragrafo precedente viene usata la definizione *automatic*, c'è da dire come prima cosa che quando hai un passaggio di un *ref*, nella funzione è obbligatorio.
In secondo luogo quello che succede senza questo prefisso è che, come per le variabili, viene allocata una parte di memoria anche per le funzioni e quindi questa è comune per ogni chiamata di tale funzione.
Questo può creare dei problemi se la stessa funzione viene chiamata in due posti diversi del testbench e crea dei risultati inaspettati.
Per esempio se ho una funzione ricorsiva è necessario utilizzarla. Qui c'è un esempio di un elevamento a potenza fatto in maniera ricorsiva che altrimenti non funzionerebbe.

```
function automatic int power_number(int number,int power);
    if(power=1) power_number=number;
    else power_number=power_number(number,power-1)*number;
endfunction
```

Viene anche utilizzata in altri contesti, infatti anche le variabili possono essere *automatic* e un esempio molto semplice è il seguente.

```
for(int i=0;i<5;i++) begin
    fork begin
        $display($sformatf("indice: %d",i));
    join
end
```

Bene, in questo caso quello che succede non è quello che ci si aspetta. Restituisce sempre "indice: 3", e questo succede perchè i è una variabile statica e nel *fork* ha sempre il valore finale. Per ovviare a questo problema bisogna utilizzare *automatic*.

```
for(int i=0;i<5;i++) begin
    automatic int k=i;
    fork begin
        $display($sformatf("indice: %d",k));
    join
end
```

Adesso vengono visualizzati tutti i numeri da 0 a 3.

2.3.3 Restituzione di output

Le funzioni possono anche restituire dei valori, direttamente o tramite l'assegnazione di una variabile. I due metodi sono intercambiabili.

```
//quando chiamo questa funzione è necessario passare una variabile di tipo int in cui viene allocato il valore della somma
function void sum_1_to_n(int number,output int sum);
    sum=number*(number+1)/2;//somma dei numeri da 1 a N
endfunction

//Viene semplicemente restituito il valore
function int sum_1_to_n(int number);
    return number*(number+1)/2;//somma dei numeri da 1 a N
endfunction
```

2.3.4 Fork...join

Certe volte è necessario fare una cosa diversa, per esempio se un segnale non cambia entro 50 picosecondi è necessario passare alla prossima istruzione. C'è una struttura che fa proprio questo: il *fork...join*.
Per la verità ci sono diverse varianti: *fork...join*,*fork...join_any* e *fork...join_none*. Ogni riga del fork è un thread a sè stante.
Qui sotto spieghiamo cosa succede nei tre casi.

```
//Qui succede che per uscire dal fork devono succedere entrambe le cose, quindi si esce quando la più lenta delle due
     istruzioni si avvera
fork begin
    @(wait_signal)
    #50ps;//aspetto un tempo pari a 50 picosecondi
join
//Adesso basta che una delle due istruzioni si avveri, cioè quella che accade prima
fork begin
    @(wait_signal)
    #50ps;//aspetto un tempo pari a 50 picosecondi
join_any

//Qui vengono eseguite entrambe e vengono creati due thread che agiscono in parallelo, però allo stesso tempo esco dal fork ed
     eseguo le istruzioni successive
```

```
fork begin
    @(wait_signal)
    #50ps;//aspetto un tempo pari a 50 picosecondi
join_none
```

2.3.5 Macro

Le macro sono un'alternativa alle funzioni e consentono di risolvere dei problemi che pone il System Verilog.
Uno di questi è quello di utilizzare un segnale *ref* all'interno di un *fork...join*, questa è un'operazione vietata, però utilizzando le macro si può passare il segnale senza usare un *ref*.
Un problema molto simile è quello che gli array devono avere la dimensione definita a priori e non possono avere gli estremi definiti in compilazione, questo si risolve allo stesso modo con una macro(viene anche utilizzato più avanti nel progetto quando è necessario monitorare un segnale senza sapere a priori quanti bit abbia questo segnale).

```
`define MONITOR_N_BIT_BUS(SIGNAL) \
  forever begin \ \\fine riga
    @(vif.``SIGNAL'') begin\

      //do something
   end \
  end

```

Questa è la struttura di base di una macro, come si può vedere alla fine di ogni riga è necessario mettere una sbarra che segnala al compilatore che è finita la riga.
Quando viene chiamata la macro si passa il nome del segnale, che in questo caso si trova all'interno di *vif* (una interfaccia connessa al dispositivo), e poi si eseguono delle operazioni.
Se volessi fare la stessa cosa per una funzione dovrei scriverne una per ogni segnale: cioè con due,tre,quattro bit ecc...
Una cosa importante su cui stare attenti è che quando viene richiamata la macro il compilatore interpreta gli ingressi che vengono passati come stringhe, fare attenzioni quindi ad usare stratagemmi con le espressioni condizionali che non vengono identificate come tali ma come stringhe.

```
bit is_valid=1;
`MONITOR_N_BIT_BUS((is_valid)?enable_h_o:reset_h_o)
```

Putroppo l'istruzione sopra non dà il risultato atteso, per evitare problema la soluzione è quella presentata qui sotto.

```
bit is_valid=1;
if(is_valid) `MONITOR_N_BIT_BUS(enable_h_o)
else `MONITOR_N_BIT_BUS(reset_h_o)
end
```

2.4 Randomizzazione

Questa è una delle funzioni più importanti per quanto riguarda la verifica digitale perchè si può automatizzare i test semplicemente chiamando una funzione che in automatico fa la randomizzazione di tutte le variabili all'interno di una classe.

2.4.1 Differenze fra rand e randc

Per fare in modo che una variabile venga randomizzata quando è chiamata l'appropriata funzione è necessario attribuirle la parola *rand* o *randc*.
La differenza è che *rand* fa semplicemente una randomizzazione casuale senza tenere in considerazione i valori passati, mentre *randc* prima di ripresentare lo stesso valore passa in rassegna tutti gli altri.
Ci sono delle restrizioni per l'uso di quest'ultima che adesso vedremo. Se ci pensiamo perchè *randc* funzioni bisogna in qualche modo "memorizzare" quello che è successo nelle precedenti iterazioni.
Facciamo un esempio così chiariamo cosa è possibile fare o meno. Supponiamo di avere un oggetto(classe) che chiamiamo *Item*, al cui interno sono presenti degli attributi che definiremo casuali.

```
1  class Item;
2      bit [2:0] number_of_tries;//Suppongo che continuo a mandarlo finchè non ricevo un "ok" della sua ricezione
3      rand int value;//il valore dell'item
4      rand [8:0] addr;
5      rand bit can_send;//mi dice se posso mandarlo
6      function void new();
7          number_of_tries=3;
8      endfunction
9  endclass
10
11 module test;
12     Item pkt=new();
13     bit is_done=0;
14     while(!is_done) begin
15         if(pkt.randomize()) begin//restituisce uno se va bene
16             //aspetta l'"ok"
17             if(is_done) break;//esci dal ciclo
18             else begin
19                 $display("Qualcosa è andato storto.");
20                 break;
21             end
22         end
23     end
24 endmodule
25
```

Con questa struttura è possibile utilizzare anche *randc* perchè faccio la randomizzazione sempre dello stesso oggetto e il compilatore implicitamente salva cosa succede prima.
Molte volte viene usata un'altra funzione che fa parte della libreria UVM(che vedremo nel capitolo successivo) però purtroppo non è compatibile con *randc*.

2.4.2 Constraints

Ci sono varie parole chiave che permettono di restringere il campo dei valori delle variabili randomiche.

Molte volte le variabili sono fra loro dipendenti riguardo ai valori che possono assumere. Bisogna fare particolare attenzione che se decido di utilizzare una variabile *randc*, questa richiede di avere la priorità rispetto alle altre. Se questo non viene rispettato il compilatore darà errore.
Facciamo un esempio modificando il codice scritto in precedenza. Ci sono due strade per imporre delle restrizioni: si può modificare il codice di partenza oppure si può utilizzare *randomize with*, facciamo un esempio e come prima soluzione modifichiamo il codice precedente.
Da notare che se è solamente necessario aggiungere delle limitazioni basta estendere la classe, però in questo caso sono state applicate delle modifiche quindi risulta più semplice riscrivere la classe.

```
1  class Item_constraint;
2      bit [2:0] number_of_tries;//Suppongo che continuo a mandarlo finchè non ricevo un "ok" della sua ricezione
3      randc int value;//utilizziamo randc
4      rand [8:0] addr;
5      bit can_send;// gli dò un valore per semplicità
6      constraint priority{solve value before addr;};
7      constraint range{addr inside{[18:97]};};// vengono utilizzati solo i valori in questo range
8      constraint specific{if(value==0) addr==1;};
9      function void new();
10         number_of_tries=3;
11         can_send=1;
12     endfunction
13 endclass
14
```

La parte di test non è stata ripetuta perchè rimane invariata, da notare che quando viene fatta un'assegnazione nelle *costraint* si deve utilizzare un doppio uguale.
Notare che ogni volta che si inserisce una limitazione è necessario utilizzare un punto e virgola.

La stessa cosa si può fare nel modo seguente lasciando invariata la classe e modificando solamente il test.

```
module test;
    Item pkt=new();
    bit is_done=0;
    while(!is_done) begin
        if(pkt.randomize() with{
            solve value before addr;
            can_send==1;
            addr inside{[18:97]};
            if(value==0) addr==1;
        }) begin//restituisce uno se va bene
            //aspetta l'"ok"
            if(is_done) break;//esci dal ciclo
            else begin
                $display("Qualcosa è andato storto.");
                break;
            end
        end
    end
endmodule

```

Si può anche impostare come distribuire la probabilità dei valori permessi. Per questo viene utilizza la parola chiave *dist*.

```
class Memory_addr
    rand bit [3:0] addr;
    rand int value ;
    constraint probability{addr dist{0:=10,[1:15]:=90}};
    constraint probability_2{addr dist{0:/10,[1:15]:/90}};
endclass

```

I due casi si differenziano così: il primo(*probability*) ha probabilità x/N,dove x è il peso specifico di ogni indirizzo, e N è il totale(in questo caso 15*90+10).
Per quanto riguarda il secondo(*probability_2*), si parla di probabilità effettiva però va divisa per il numero di elementi. Questo vuol dire che lo zero ha probabilità del 10% e ogni altro numero del 6%.
Se per un test specifico devo eliminare qualche constraint non è ovviamente necessario modificare tutto il codice sorgente, ma basta semplicemente disabilitare quella non necessaria.

```
module test;
    Memory mem=new();
    //Questo disabilita la constraint probability
    //se è abilitata
    if(mem.probability.constraint_mode()==1)
        mem.probability.constraint_mode(0);
    end
    if(!mem.randomize()) $display("Qualcosa è andato storto.");
endmodule

```

2.5 Comunicazione fra processi

Molto spesso è necessario che quando succede un evento vengano successivamente eseguite delle operazioni.

2.5.1 Eventi

Gli eventi ci vengono in aiuto quando è necessario comunicare fra diversi blocchi di uno stesso modulo. Di seguito un esempio dimostrativo:

```
module event_handling;
    event eve;
    fork
        begin
            eve=new();
            #20ps;
            ->eve;
        end
        begin
            @(eve.triggered);
            //in alternativa
            //wait(eve.triggered);
            $display($sformatf("Sono passati %d secondi dall'inizio della simulazione",$time));
        end
    join
endmodule
```

Se l'evento viene lanciato nello stesso momento in cui ci mettiamo in ascolto, non viene visto e si dovrebbe invece ricorrere all'uso della parola chiave *wait*.

2.5.2 Mailbox

Una struttura molto utile è quella delle *mailbox*, infatti se ho due parti di uno stesso modulo e voglio inviare delle informazioni però queste parti non sono sincrone posso creare una gestione basata sull'uso di eventi e *mailbox*.
Supponiamo che ho due parti, una che invia dati ogni due nanosecondi e un'altra che li mostra su un display ogni tre nanosecondi.
Le *mailbox* quando vengono create richiedono quanti oggetti possono contenere, ovviamente non è possibile fornirgli più oggetti quindi bisogna in qualche modo "liberare" un posto.
Per semplicità se la *mailbox* è piena, aspetto quattro nanosecondi così sono sicuro che si sia liberato un posto.

```
module event_handling;
    event eve;
    mailbox box=new(3);//al massimo tre oggetti
    fork
        begin
            automatic int i=0;
            while(i<8) begin
                if(box.try_put(i)) #2ns;
                else #4ns
                end
                i++;
            end
        end
        forever begin
            #3ns;
            ->eve;
        end
        forever begin
            @(eve.triggered);
            //in alternativa
            //wait(eve.triggered);
            automatic int k;
            if(box.try_get(k)) $display($sformatf("Oggetto numero: %d",k));
            else $display("Qualcosa è andato storto");
            end
        end
    join
endmodule
```

3

Libreria UVM

3.1 Introduzione

Nel capitolo precedente abbiamo visto le funzionalità principali del System Verilog e di come si potrebbe imbastire un ambiente di test anche senza utilizzare quello che vedremo in questo capitolo.
La libreria UVM è un'estensione del System Verilog, infatti è possibile importarla ma non è necessario usarla, e non è un ambiente a sè stante.

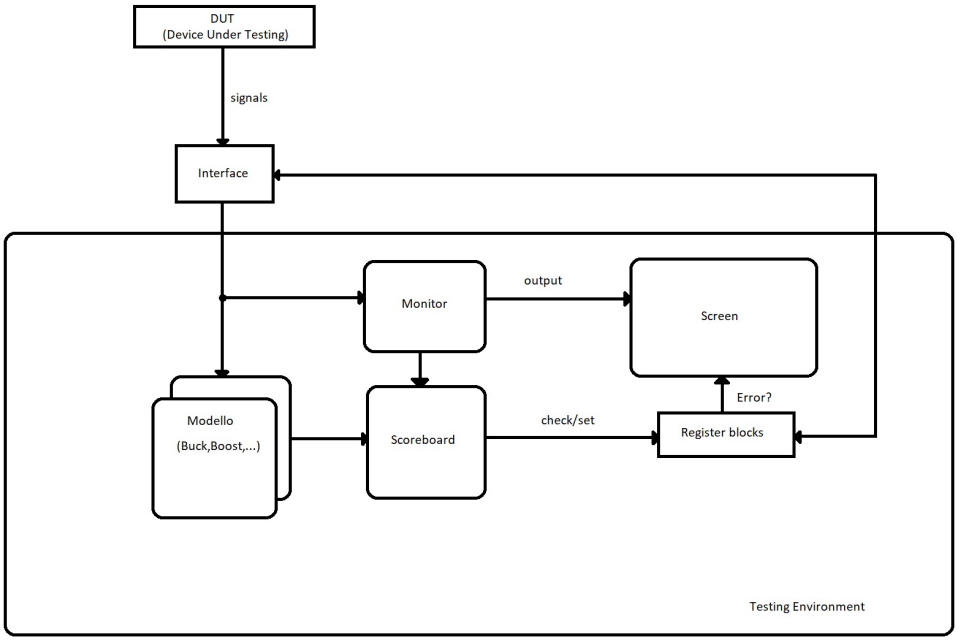

Figura 3.1: Schema di un ambiente di test

La particolarità di questa libreria, e il motivo per cui è diventata così utilizzata, risiede nel fatto che non solo fornisce già una struttura base (ovviamente deve essere "riempita") ma anche delle funzionalità non poco utili che vedremo più in dettaglio in questo capitolo.
Partiamo da una visione d'insieme e vediamo quali classi sono fornite.
Come possiamo vedere ci sono vari blocchi, alcuni sono forniti dalla libreria UVM, come lo scoreboard ed il monitor, ed altri invece vanno creati da zero, come i modelli.
Poi ci sono delle classi che non sono nemmeno visibili in questo schema ma che verranno fuori quando analizzeremo in dettaglio dei blocchi specifici.
Inoltre questi blocchi devono scambiare informazioni in qualche modo ed useremo varie strutture per renderlo possibile.
Volevo precisare che quando viene costruito un test, in generale si testa un ambiente a segnali misti, analogici e digitali, generalmente viene fornito un modello per i segnali analogici e si ha invece una versione RTL per la parte digitale.
Quest'ultima parte che di solito è realizzata in System Verilog, o in alternativa in VHDL, pone delle limitazioni sulle istruzioni che possono essere usate perchè poi c'è un processo di sentetizzazione.
Queste limitazioni non sono imposte anche all'ambiente di test che può utilizzare qualsiasi struttura perchè è come un'impalcatura fuori da un edificio che permette solamente di vedere che stia in piedi.
In questo capitolo partiremo dall'inizio della catena, che sono gli oggetti, fino ad arrivare all'ambiente che contiene tutto il test, parlando anche di come è possibile connettere e scambiare informazioni non solo fra i diversi blocchi, ma anche a livello di ambiente di test.

3.2 Classi base

Ci sono due classi base: *uvm_object* e *uvm_component*. La prima è utilizzata per qualsiasi oggetto che vogliamo passare tramite l'ambiente di verifica, non è necessario ma è altamente consigliato perchè ci sono dei benefici che vedremo in seguito.
La seconda invece serve per modellare delle parti dell'ambiente di test (Boost, Buck, ecc...) ed è anche la classe da cui tutte le classi base estendono.

3.2.1 Gli oggetti

Gli oggetti sono delle classi che vengono fornite dalla libreria UVM e che ereditano dei metodi di *copy*, *convert2string* e *clone* che vanno definiti in fase di estensione.

```
1  import uvm_pkg::*;//necessario altrimenti non funziona nulla
2
3  class Item extends uvm_object;
4
5      `uvm_object_utils(Item) //necessario vedremo dopo a cosa serve
6
7      int value;
8      int addr;
9
10     function void new(string name="item");
11         super.new(name);
12     endfunction
13
14     virtual function uvm_object clone();
15         uvm_object obj=Item::type_id::create(); //factory create può essere sostituito da new()
16         obj.value=this.value;
17         obj.addr=this.addr;
18         return obj;
19     endfunction
20
21     virtual function string convert2string();
22         string s= $sformatf("Address: %d , Value: %d",addr,value);
23         return s;
24     endfunction
25
26     virtual function bit compare(uvm_object obj);
27         //Questo può essere definito in un altro modo, per esempio mi può interessare solo che il campo value sia lo stesso
28         if(obj.addr==this.addr && obj.value==this.value) return 1;
29         return 0;
30     endfunction
31 endclass
```

Questo è un modo per farlo, abbastanza intuitivo, ed è la stessa cosa che si fà nella programmazione ad aggetti.
La parola chiave *virtual* permette, se ci sono classi che estendono *Item*, di utilizzare un concetto noto come polimorfismo che vuol dire che quando viene chiamato questo metodo da un oggetto che è stato incapsulato in un oggetto di questo tipo viene chiamato il metodo relativo al tipo specifico contenuto nell'oggetto.
La libreria UVM permette un'alternativa molto semplice e veloce che non richiede la creazione di nessuna funzione.

```
import uvm_pkg::*;//necessario altrimenti non funziona nulla

class Item extends uvm_object;

    //definendo queste macro i metodi print, clone e compare vengono automaticamente creati e se specifico diveramente posso
      anche non includere certi parametri in alcuni metodo(come per esempio compare)
    `uvm_object_utils_begin(Item) //necessario vedremo dopo a      `uvm_field_int(value, UVM_DEFAULT)
        `uvm_field_int(addr, UVM_DEFAULT)
        `uvm_field_string(name, UVM_NOCOMPARE)
    `uvm_object_utils_end(Item)

    int value;
    int addr;
    string name;

    //questa funzione è sempre necessario definirla
    function void new(string name="item");
        super.new(name);
    endfunction

endclass
```

3.2.2 I COMPONENTI

Per quanto riguarda i componenti, questi fornisco delle funzioni di base che vengono utilizzate in momenti diversi della loro vita.

```
import uvm_pkg::*;//necessario altrimenti non funziona nulla

class Boost extends uvm_component;
    //E' sempre necessaria solo che adesso parliamo di componenti
    `uvm_component_utils(Boost)
    //è necessario definirla sempre
    function void new(string name="boost",uvm_component parent=null);
        super.new(name,parent);
    endfunction
    //extern significa solamente che bisogna definire cosa fa questa funzione al dì fuori della classe
    extern function void build(uvm_phase phase);

    extern function void connect(uvm_phase phase);

    extern function void run(uvm_phase phase);
endclass
function void Boost::build(uvm_phase phase);
    super.build(phase);
endfunction
```

3.2.3 LE FASI DI UN COMPONENTE

Parlando dei componenti si deve parlare anche di fasi perchè sono molto importanti e sono saltate fuori nel paragrafo precedente.

Nell'immagine qui sopra si possono vedere le tre fasi principali di un componente, ce ne sarebbero altre ma per semplicità elenchiamo solo queste perchè sono le più usate. Partiamo dalla fase di build che, come il nome può fare intuire, è quella in cui vengono instanziati i componenti.

Se c'è qualche variabile che deve essere usata bisogna ricordarsi di crearla qui, in questa fase vengono anche caricate le variabili di ambiente come vedremo nella sezione successiva.

La fase di connect serve a connettere i vari moduli di un testbech perchè non possiamo sapere a priori che la fase di build di un altro componente sia finita, causando magari un' eccezione perchè chiamo un componente non istanziato, ed è quindi necessario fare

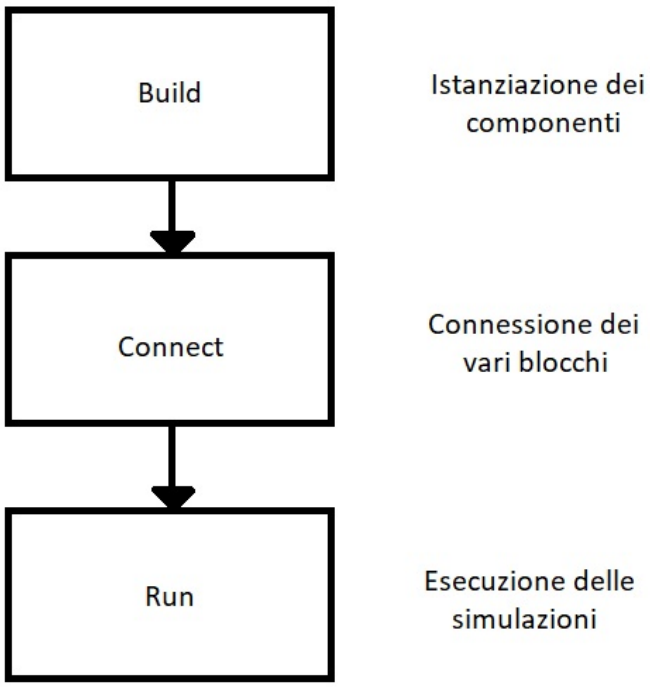

Figura 3.2: Fasi di un componente

queste operazioni in un frame temporale in cui si è certi che non succeda una cosa del genere.
Nella fase di run vengono lanciate tutte le simulazioni e ovviamente si esce, se non ci sono stati errori, quando tutti i processi sono terminati.

3.3 Organizzazione dell'ambiente di test

Come accennato in precedenza in questa sezione verrà esaminata in maniera approfondita la struttura di un ambiente di verifica.
Partiamo dalle sequenze, esaminando successivamente il monitor, il driver, il sequencer e l'agent che verranno esaminati singolarmente, che però solitamente si raggruppano assieme perchè si possono considerare un singolo blocco.
Successivamente vediamo lo scoreboard, che di solito è dove vengono gestiti i registri ed infatti andremo ad analizzarli in quell'istanza, l'environment ed il test.
Tutte le parti menzionate nel paragrafo precedente sono classi di base della libreria UVM ed ognuna estende o gli *uvm_object* o l'*uvm_component*.
Tutte queste classi in qualche modo vanno "collegate" e per questo impareremo a conoscere le porte ed il loro funzionamento, però introdurremo anche altre strutture che permettono di scambiare dati fra questi moduli.

3.3.1 Le sequenze

Le sequenze estendono la classe *uvm_object* e contengono una serie di istruzioni al loro interno che vanno a sollecitare il DUT (Device Under Testing).

```
1
2  class Item extends uvm_object;
3
4      `uvm_object_utils_begin(Item) //necessario vedremo dopo a      `uvm_field_int(value, UVM_DEFAULT)
5          `uvm_field_int(addr, UVM_DEFAULT)
6          `uvm_field_string(name, UVM_NOCOMPARE)
7      `uvm_object_utils_end(Item)
8
9      rand int value;
10     rand int addr;
11
12     //questa funzione è sempre necessario definirla
13     function void new(string name="item");
14         super.new(name);
15     endfunction
16
17 endclass
18
19 class my_sequencer extends uvm_sequencer;
20     `uvm_component_utils(my_sequencer)
21
22     function void new(string name="my_sequencer",uvm_component parent=null);
23         super.new(name,parent);
24     endfunction
25 endclass
26
27 //dichiaro una sequenza che ha un tipo Item
28 class my_seq extends uvm_sequence#(Item);
29     //come abbiamo già visto è necessaria
30     `uvm_object_utils(my_seq)
31     //questo è necessario perchè bisogna dire alla sequenza quale è il sequencer che la analizza, generalmente non è
         necessario crearne uno ma basta usare quello fornito dalla libreria UVM
32     `uvm_declare_p_sequencer(my_sequencer)
33
34     Item data;
35
36     function new(string name="my_sequence");
37         super.new(name);
38     endfunction
39
40     virtual task pre_start();
41      if(starting_phase!=null)
42         starting_phase.raise_objection(this);
43     endtask
44
45     virtual task body();
46         data=Item::type_id::create("item");
47
48         //Questo potrebbe essere all'interno in un ciclo che viene ripetuto più volte
49
50         start_item(data);
51         if(!data.randomize()) `uvm_fatal(get_type_name(),"Qualcosa è andato storto.");
52         finish_item(data);
53     endtask
54
55     virtual task post_start();
56      if(starting_phase!=null)
57         starting_phase.drop_objection(this);
58     endtask
59 endclass
```

Con questo codice la sequenza viene interpretata dal sequencer e mandata al driver che vedremo nella prossima sezione.
Se non viene chiamata *raise_objection(this)* il test finisce prima che la sequenza sia terminata, alla fine viene chiamata *drop_objection(this)* per far presente al test che è terminata la sequenza.
Molte volte però succede che all'interno di una sequenza devo lanciare altre sequenze,

per esempio supponiamo che questa è una sequenza per il Boost però prima di poterla mandare ci deve essere una sequenza di startup del dispositivo nel complesso.

```
virtual task body();
    //sequenza di startup
    startup_sequence start_seq=startup_sequence::type_id::create("start_seq");
    if(!start_seq.randomize()) `uvm_fatal(get_type_name(),"Qualcosa è andato storto.");
    //start_sequencer è il tipo del sequencer
    start(.sequencer(start_sequencer),.parent_sequence(this));
    data=Item::type_id::create("item");

    //Questo potrebbe essere all'interno in un ciclo che viene ripetuto più volte

    start_item(data);
    if(!data.randomize()) `uvm_fatal(get_type_name(),"Qualcosa è andato storto.");
    finish_item(data);
endtask
```

3.3.2 Driver

La classe di driver, come abbiamo detto prima, viene collegata alle sequenze dal sequencer e il suo compito è di fare da tramite fra l'ambiente di test e il DUT.

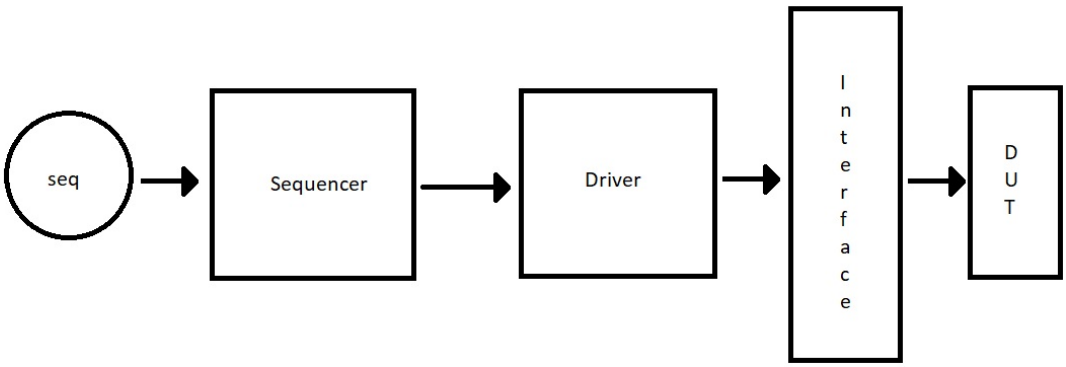

Figura 3.3: Schema logico

Ricolleghiamoci a quello scritto nella sezione precedente e supponiamo che il Sequencer invii una sequenza di tipo *my_seq* al driver, lasciando perdere la sequenza di startup che complica solamente le cose, e vediamo come va gestita.
La classe driver contiene una porta con la quale dialoga con il sequencer, vedremo poi come fare tutti i settaggi necessari, intanto basta sapere che è un oggetto di tipo *uvm_seq_item_pull_port* e si chiama *seq_item_port*.

```
import uvm_pkg::*;

class my_driver extends uvm_driver;
    `uvm_component_utils(my_driver)

    function void new(string name="my_driver",uvm_component parent=null);
        super.new(name,parent);
    endfunction

```

```
10      function build_phase(uvm_phase phase);
11          super.build_phase(phase);
12      endfunction
13
14      virtual task run_phase(uvm_phase phase);
15          super.run_phase(phase);
16          Item req;
17          forever begin
18              //mette il risultato nella variabile req
19              seq_item_port.get_next_item(req);
20              //una alternativa è seq_item_port.try_next_item(req) che restituisce un uno se c'è un item
21
22              //fai delle elaborazioni
23              drive_item(req);
24
25              //avvisa il sequencer che questo item è stato processato
26              seq_item_port.item_done()
27          end
28
29      endtask
30
31      task void drive_item( Item item);
32          //per esempio setti un segnale dell'interfaccia
33      endtask
34  endclass
```

Il codice sopra resta semplicemente in attesa di un item che viene mandato dal Sequencer e quando arriva un item lo elabora e converte il dato in istruzioni per "guidare" il DUT tramite la sua interfaccia.

3.3.3 Il monitor

Il compito del monitor è quello di monitorare dei segnali e generalmente viene connesso all'interfaccia e controlla che non succedono cose inaspettate e manda degli oggetti allo scoreboard tramite una porta.

Le porte

Le porte sono molto utili per scambiare dati e ci sono parecchie varianti che funzionano in maniera leggermente diversa.
In questa sezione analizziamo semplicemente un sistema che collega un monitor ad

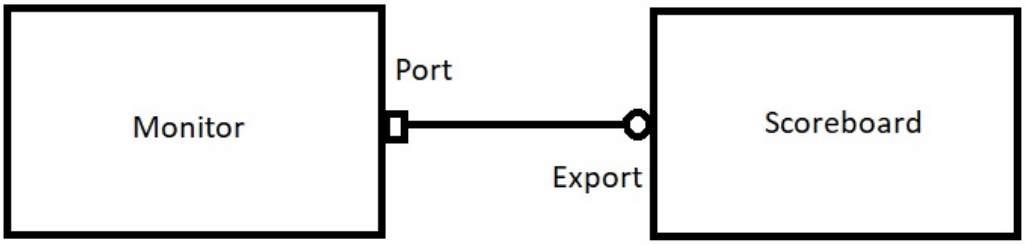

Figura 3.4: Connessione fra componenti

uno Scoreboard tramite una *uvm_analysis_port* ed il dato viene ricevuto da una *uvm_analysis_imp*.

```
class my_monitor extends uvm_monitor;
    uvm_analysis_port#(Item) monitor_port;

    //...

    function void build_phase(uvm_phase);
        super.build_phase(phase);
        monitor_port=new("mon_port",this);
    endfunction

    task void run_phase(uvm_phase phase);
        super.run_phase(phase);
        Item item;
        if(!item.randomize()) `uvm_fatal(get_type_name(),"Qualcosa è andato storto.");
        //mando un item a tutti i subscriber di questa porta
        monitor_port.write(item);
    endtask
endclass
```

3.3.4 Scoreboard

Lo Scoreboard in genere fa delle verifiche sui registri quando gli arriva un Item dal monitor e controlla che i dati ricevuti coincidano con quello che è salvato in memoria.

```
class Scoreboard extends uvm_scoreboard;
    uvm_analysis_imp#(Item,my_scoreboard) mon_export;

    //...

    function void build_phase(uvm_phase);
        super.build_phase(phase);
        mon_export=new("mon_export",this);
    endfunction

    function void write(Item item);
        $display("E' arrivato un item.");
    endfunction
endclass
```

Lo scoreboard viene anche utlizzato per gestire la macchina a stati finiti.

3.3.5 I registri

I registri sono una parte molto importante di un test perchè permettono di avere una copia di quello che è effettivamente salvato sul dispositivo che stiamo testando. Solitamente sono dentro una classe che viene passata e vengono poi passati dall'ambiente allo scoreboard tramite delle variabili. Come possiamo notare dalla figura, il registro effettivo e quello creato da noi non hanno lo stesso valore e spesso è questo il caso perchè il valore va aggiornato, "mirrored", in certi casi.
Vediamo adesso come istanziare effettivamente i registri e quali sono le operazioni che posso effettuare su di essi.

```
class my_reg extends uvm_reg;
    rand uvm_reg_field enable;

    function void new(string name="my_reg");
        super.new(name,16,UVM_NO_COVERAGE);//16 bit
    endfunction

```

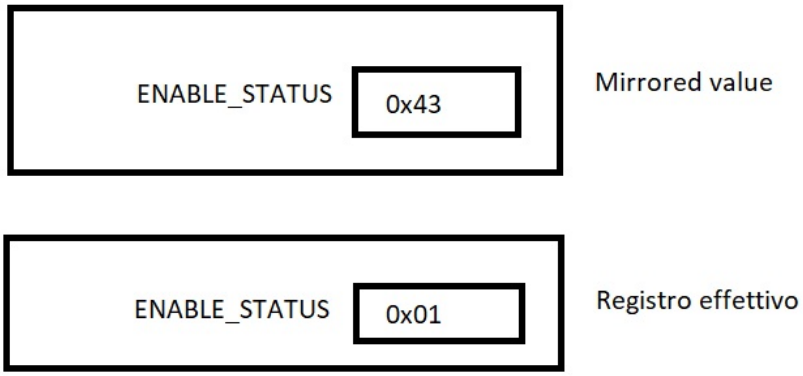

Figura 3.5: Registri

```
8     virtual function void build();
9         enable=uvm_reg_field::type_id::create("enable");
10
11        //enable.configure(parent,size,lsb_pos,access,volatile,reset,has_reset,is_rand,individually_accessible);
12        //questo registro ha dimensione uno ed è accessibile singolarmente in lettura e scrittura, può anche essere resettato
      scrivendo un zero logico
13        enable.configure(this,1,1,"RW",0,0,1,1,1);
14    endfunction
15 endclass
16
17 class my_reg_block extends uvm_reg_block;
18    rand my_reg en;
19
20    function void new(string name="my_reg_block");
21        super.new(name,UVM_NO_COVERAGE);
22    endfunction
23
24    virtual function void build();
25        this.default_map=create_map("",0,4,UVM_LITTLE_ENDIAN,0);
26        //creo il registro
27        en=my_reg::type_id::create("my_register");
28
29        //configuro il registro
30        en.configure(this,null,"");
31
32        //costruisco il registro
33        en.build();
34
35        //aggiungo il registro alla default_map
36        this.defalut_map.add_reg(en,'UVM_REGISTER_WIDTH'h0,"RW",0);
37    endfunction
38 endclass
```

Questo è il codice per la creazione di un registro ed è poi possibile fare delle operazioni su questi registri, generalmente di lettura e scrittura stando attenti che venga tenuto aggiornato al valore presente in memoria.

3.3.6 AGENT

L'agente è una classe che contiene il monitor,il driver e il sequencer e crea il collegamenti fra di loro. Vediamo adesso cosa è necessario fare nella classe dell'agent, generalmente si istanziano i componenti e si creano i collegamenti.

```
1 class my_agent extends uvm_agent;
```

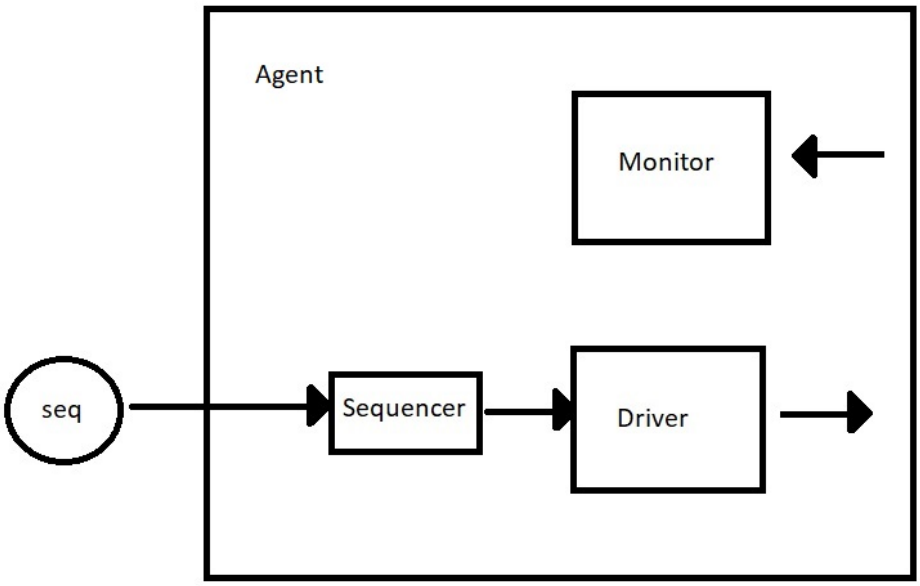

Figura 3.6: Agent

```
2     `uvm_component_utils(my_agent)
3
4     my_driver driver;
5     my_monitor monitor;
6     my_sequencer my_seqr;
7
8     function new(string name="my_agent",uvm_component parent=null);
9         super.new(name,parent);
10    endfunction
11
12    virtual function void build_phase(uvm_phase phase);
13        super.build_phase(phase);
14        //inizializzo i componenti
15        //se è attivo questo agent
16        if(get_is_active()) begin
17            driver=my_driver::type_id::create("my_driver");
18            my_seqr=my_sequencer::type_id::create("my_sequencer");
19        end
20        //questo è sempre presente
21        monitor=my_monitor::type_id::create("my_monitor");
22    endfunction
23
24    virtual function void connect_phase(uvm_phase phase);
25        super.connect_phase(phase);
26        //connetto il sequencer al driver
27        driver.seq_item_port.connect(my_seqr.seq_item_export);
28    endfunction
29 endclass
```

3.3.7 Environment

L'ambiente contiene una serie di blocchi che devono essere collegati fra loro, per esempio posso avere un agent che viene collegato con uno scoreboard.

```
class my_env extends uvm_environment;
    `uvm_component_utils(my_env)

    my_agent agent;
    my_scoreboard scoreboard;

    function new(string name="my_agent",uvm_component parent=null);
        super.new(name,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        //inizializzo i componenti

        agent=my_agent::type_id::create("my_agent");
        scoreboard=my_scoreboard::type_id::create("my_scoreboard");
    endfunction

    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        //connetto lo scoreboard con l'agent, per la precisione con il monitor
        agent.monitor.monitor_port.connect(scoreboard.mon_export);
    endfunction
endclass
```

3.3.8 Test

Il test è l'involucro che comprende tutto l'ambiente di sviluppo e solitamente ci sono vari ambienti: digitale, analogico, ecc...
Questi ambienti devono essere inizializzati e connessi fra di loro se necessario e anche con il DUT attraverso l'interfaccia che solitamente viene settata e "trasportata" nelle classi più interne per fare un "driving" e "monitoring" dei segnali.

```
class my_test extends uvm_test;
    `uvm_component_utils(my_test)

    my_env env;

    function new(string name="my_agent",uvm_component parent=null);
        super.new(name,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        //inizializzo i componenti

        env=my_env::type_id::create("my_env");
    endfunction

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        my_seq seq=my_seq::type_id::create("my_seq");

        phase.raise_objection(this);

        //dò inizio alla simulazione
        seq.start(env.agent.my_seqr);

        phase.drop_objection(this);
    endtask
endclass
```

3.4 Gestione degli eventi ed oggetti

La libreria UVM ha delle funzionalità molto importanti che semplificano molto il lavoro.

3.4.1 UVM FACTORY

Incominciamo ritornando a quello che è stato detto in precedenza riguardo agli oggetti e i componenti: c'era un'istruzione che andava assolutamente inserita.

```
//per i componenti
`uvm_component_utils(Boost)
//per gli oggetti
`uvm_object_utils(Item)
```

Quello che questa riga di codice fa è registrare presso la UVM factory un oggetto che poi verrà instanziato usando il metodo create.

```
Item base_item=Item::type_id::create();
```

Questo processo di registrazione nella factory dà la possibilità di apportare delle modifiche agli oggetti senza la necessità di modificare tutto il codice sorgente.
Per esempio ho un pacchetto che mi arriva ed è cambiato il contenuto che viene spedito posso creare un oggetto nuovo, che deve necessariamente estendere quello che vai a sostituire, e posso usare il metodo di override per sostuire i due oggetti.

```

class obj_1 extends uvm_obj;
    `uvm_component_utils(obj_1)

    function void new(string name="obj_1");
        super.new(name);
    endfunction
enclass

class obj_2 extends obj_1;
    `uvm_component_utils(obj_2)

    function void new(string name="obj_2");
        super.new(name);
    endfunction
enclass

class env extends uvm_env;
    `uvm_component_utils(env)
    obj_1 pkt;
    function void new(string name="env",uvm_parent=null);
        super.new(name,parent);
        pkt=obj_1::type_id::create("obj_1");
    endfunction
enclass

class test extends uvm_test;
    `uvm_component_utils(test)
    env test_env;
    function void new(string name="test",uvm_parent=null);
        super.new(name,parent);
    endfunction

    function void build(uvm_phase phase);
        super.build(phase);
        test_env::type_id::create("env_0");
        //abbiamo un link alla factory
        uvm_factory factory= uvm_factory::get();
        //sostituisce ogni oggetto di tipo obj_1 con obj_2
        //ovviamente se non uso la libreria UVM e non li registro questa cosa è inutile
        set_type_override_by_type(obj_1::get_type(),obj_2::get_type());
        //Questo sostituisce solo le variabili all'interno di test_env
        //set_type_override_by_type("test_env.*",obj_1::get_type(),obj_2::get_type());
        //si può fare la stessa cosa usando
        //set_type_override_by_name("obj_1","obj_2",{get_full_name(),".test_env.*"});
        //{get_full_name(),".test_env.*"} è l'operatore concatenazione fra il percorso completo e la variabile test_env
        //da qui è possibile vedere se ci sono stati degli override e quali oggetti sono registrati
        factory.print();
    endfunction

    function void run(uvm_phase phase);
        super.run(phase);
```

```
53     endfunction
54 enclass
```

Nel codice sopra viene creato un ambiente di test con all'interno un ambiente che contiene una variabile registrata nella factory.
Nella fase di build viene poi fatta una override alla factory scambiando il tipo *obj_1* per il tipo *obj_2* e vengono forniti varie soluzioni che sono tra loro equivalenti e a discrezione di chi le usa.
Prima si era creata una handle alla factory che viene utilizzata per stampare la configurazione ed è possibile vedere gli oggetti registrati e anche se ci sono stati degli override su che oggetti sono stati fatti.

3.4.2 UVM CONFIGURATION DATABASE

Una funzionalità utile è quella del configuration database che è un oggetto che permette di salvare delle variabili in memoria ed accedervi in un secondo momento.

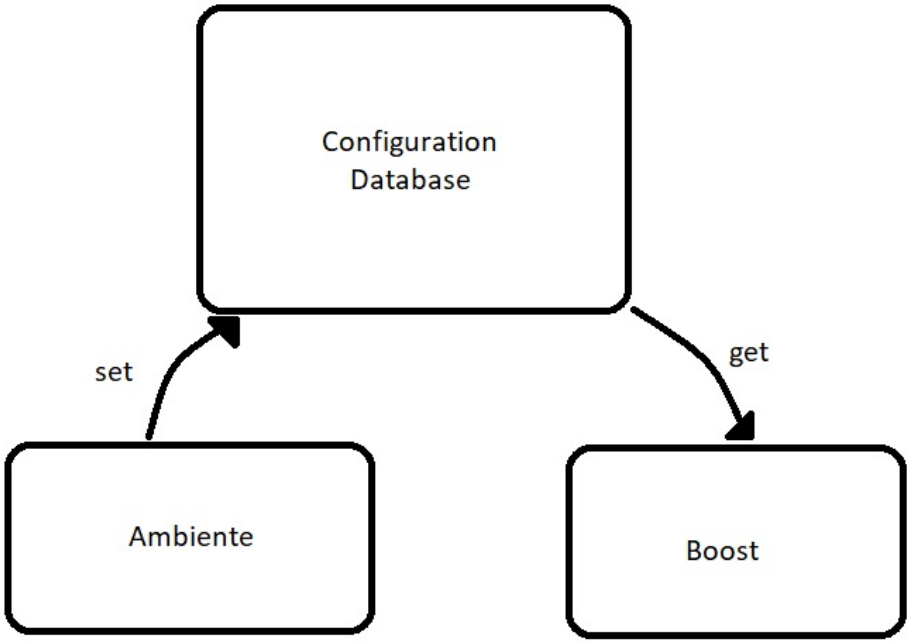

Figura 3.7: Database di configurazione

E' particolarmente importante perchè queste variabili vengono settate a livello di ambiente e non del singolo modulo e possono essere utilizzate per scambiare oggetti di configurazione e/o interfacce.
Supponiamo che abbiamo un modulo che necessita di alcune variabili di ambiente, magari definite in una classe, e non possono avere un valore deciso a priori ma ad un certo punto vengono inizializzate devo avere un modo per passarle e qui entra in gioco il database di configurazione.
Di solito queste variabili vengono settate e/o lette nella fase di build di un componente.
Supponiamo che abbiamo un ambiente che setta una variabile, per semplicità un intero che rappresenta quanti secondi aspettare dopo l'accensione prima di continuare con la simualazione, e il Boost che legge questa variabile.

```
//supponiamo che la variabile boost_m è all'interno di Environment
function Environment::build_phase(uvm_phase phase);
    super.build_phase(phase);
    //uvm_config_db::set#(int)(context,path,name,value);
    //se context è null ha il valore path altrimenti è {context,".",path} ricordando che questo è l'operatore concatenazione
    uvm_config_db::set#(int)(null,"boost_m","wait_time",3);
endfunction

function Boost::build_phase(uvm_phase phase);
    super.build_phase(phase);
    //wait_time è una variabile di tipo int definita in Boost
    if(!uvm_config_db::get#(int)(this,"*","wait_time",wait_time))
        `uvm_fatal(get_type_name(),"Non esiste questa variabile.");
endfunction
```

Si faccia un controllo che questa variabile venga effettivamente caricata in fase di build. Nel mio codice è stato inserito un messaggio di errore che ferma la simulazione se questa variabile non esiste, si può anche fare diversamente utilizzando invece di *'uvm_fatal(...)* un *'uvm_info(...)* o *'uvm_error(...)* che non interrompono la simulazione.
Questo però è controproducente perchè magari la variabile non è sempre usata e la simulazione in questo caso funziona ma non in altri, è meglio quindi saperlo e trovare il motivo fin da subito.

3.4.3 UVM POOL ED EVENTI

Molto spesso è necessario in un ambiente di test avere un dialogo fra moduli diversi, supponiamo di avere due modelli: uno è quello di un Boost e l'altro è uno Scoreboard. Succede un evento che per esempio fa in modo che il boost si spenga. Quello che voglio che succeda è che lo Scoreboard venga messo al corrente di questa situazione e prenda le dovute precauzioni.
Per fare questo si utilizzano gli eventi, il Boost farà un "trigger" a quell'evento e ci sarà una funzione nello Scoreboard che è in ascolto per quel preciso evento e farà qualche cosa. Vediamo come può essere implementata una cosa del genere.
Supponiamo di essere per entrambi nella funzione *run()* perchè di solito è qui che gli eventi vengono lanciati e/o catturati.

```
//questa è la classe utilizzata in precedenza, viene messa qui per facilitare il lettore senza tornare indietro
```

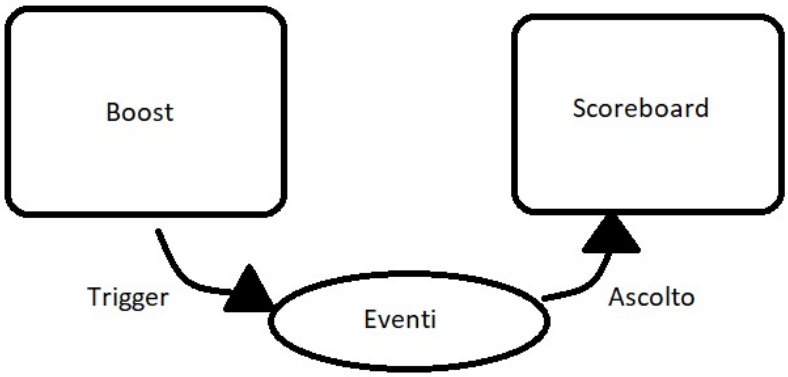

Figura 3.8: Relazione causa-effetto

```
2  class Item extends uvm_object;
3
4      //definendo queste macro i metodi print, clone e compare vengono automaticamente creati e se specifico diveramente posso
         anche non includere certi parametri in alcuni metodo(come per esempio compare)
5      `uvm_object_utils_begin(Item) //necessario vedremo dopo a      `uvm_field_int(value, UVM_DEFAULT)
6          `uvm_field_int(addr, UVM_DEFAULT)
7          `uvm_field_string(name, UVM_NOCOMPARE)
8      `uvm_object_utils_end(Item)
9
10     int value;
11     int addr;
12     string name;
13
14     //questa funzione è sempre necessario definirla
15     function void new(string name="item");
16         super.new(name);
17     endfunction
18
19 endclass
20
21 task Boost::run_phase(uvm_phase phase);
22     super.run_phase(phase);
23     uvm_event data=uvm_event_pool::get #(uvm_event#(Item))("data");
24     Item item=new("scoreboard_item");
25     item.value=12;
26     item.addr=1;
27     item.name="Boost";
28     #30ns;
29     data.trigger(item);
30 endtask
31
32 task Scoreboard::run_phase(uvm_phase phase);
33     super.run_phase(phase);
34     uvm_event data=uvm_event_pool::get #(uvm_event#(Item))("data");
35     data.wait_trigger();
36     Item item=data.get_trigger_data();
37     $display(item.print());
38 endtask
```

Nel codice il Boost crea un evento e gli associa un oggetto, Item, che contiene dei dati utili allo Scoreboard e quest'ultimo aspetta che l'evento venga lanciato.
Passati 30 nanosecondi lo Scoreboard riceve l'evento e stampa sullo schermo l'oggetto che gli è stato passato tramite l'evento.
E' necessario però fare un chiariemento: entrambi i modelli utilizzano la funzione get, che non è un errore poichè la funzione set non esiste.
La funzione get nel Boost quello che effettivamente fa è creare questo evento e nello Scoreboard invece questo evento è copiato garantendo quindi che i due modelli siano

effettivamente collegati dallo stesso evento.
Da notare che non è necessario fornire un oggetto da passare con l'evento, ma è stato introdotto nell'esempio per completezza e anche perchè può risultare una cosa molto utile da sapere.

4

La classe dei modelli

4.1 Introduzione

Questa è una classe generale, che idealmente ogni modulo di un ambiente di test dovrà estendere e fornisce una serie di funzioni e classi di base che permettono la gestione della catena dei fault.

4.2 La classe Item

Nelle sezioni successive vedremo che le classi dei modelli sono solitamente collegate ad altri moduli attraverso delle porte.
Tramite questi collegamenti è possibile trasferire un dato qualsiasi, questo oggetto rappresenta l'informazione che vogliamo spedire ed è possibile definire il tipo da passare in fase di istanziazione del modello.

```
1  class model_item #(type data_type=bit [3:0]) extends model_item_base;
2
3    data_type v_value;
4    bit value;   // si può estendere a più bit se necessario
5    uvm_check_e valid;
6    regs_cmd_t  cmd;//{PREDICT,CONF_CHECK,ALIGN2CURR}
7    int unsigned extra_time;//tempo entro il quale fare il trigger dell'interrupt(int_wait_time)
8
9    //this refers to this specific type
10   typedef model_item#(data_type) this_type;
11
12   static this_type type_handle=get_type();
13
14   static function this_type get_type();
15
16   function model_item_base get_type_handle();
17
18   function new (string name = "model_item");
19     super.new(name);
20   endfunction : new
21
22   virtual function string convert2string();
23     string s;
24     s = $sformatf("\treg: %0p\n", v_value);
25     s = {s, $sformatf("\tvalue: %0d\n", value)};
```

```
26      s = {s, $sformatf("\tvalid: %0p\n", valid)};
27    s = {s, $sformatf("\tcmd: %0p\n", cmd)};
28    s = {s, $sformatf("\textra_time: %0p\n", extra_time)};
29      return s;
30    endfunction
31
32  endclass: model_item
```

La classe sopra ha delle particolarità che spieghiamo nei paragrafi successivi, intanto precisiamo che estende una classe virtuale vuota.
La cosa più importante di questa classe è la variabile *type_handle*. Prendiamo il caso in cui ci sono vari modelli che implementano questa classe, saranno collegati tramite una porta ad un'altra entità.
Supponiamo adesso che questi oggetti arrivino a questa entità che deve in qualche modo scremarli in base al posto da cui provengono.
Questa variabile ha un tipo specifico che varia in base al tipo di dato con il quale viene istanziata e quindi è possibile capirne la provenienza controllandone il tipo.
Vediamo adesso un esempio di come questa cosa è implementata, precisando che è necessario avere a disposizione una variabile per ogni tipo di dato.

```
1  class top_soc extends uvm_component;
2
3      `uvm_component_utils(top_soc)
4
5      //dichiaro le variabili
6      //ldos_trks_regs_t,abist_t e buck_t sono variabili di tipo enumerazione
7      model_item#(ldos_trks_regs_t) ldos;
8      model_item#(abist_t) abist;
9      model_item#(buck_t) buck;
10
11     function void new(string name="top_soc",uvm_component parent=null);
12         super.new(name,parent);
13         ldos=model_item#(ldos_trks_regs_t)::get_type();
14         abist=model_item#(abist_t)::get_type();
15         buck=model_item#(buck_t)::get_type();
16     endfunction
17
18     //questa funzione restituisce una stringa che dice da che modello proviene il dato e si può anche personalizzare in base
         alle esigenze
19     function string model_type(model_item_base item);
20       string s;
21       if (item.get_type_handle()==ldos.get_type_handle())
22         s= "ldos type";
23       else if (item.get_type_handle()==abist.get_type_handle())
24         s="abist type";
25       else if (item.get_type_handle()==buck.get_type_handle())
26         s="buck type";
27       else s="undefined type";
28       return s;
29     endfunction
30 endclass
```

4.3 Segnali digitali: come modellizzarli

Un'altra funzione molto utile è fornita dalla classe *base_signal* che è composta da tre variabili: il segnale, un check che dice se bisogna controllare questo segnale (ci sono molti casi in cui è disabilitato perchè magari siamo in una zona temporale in cui il valore potrebbe essere non definito) e un altro segnale che dice se ci si aspetta un cambiamento di questo segnale.
A questi segnali sono associati sei metodi di set e get per le tre variabili ed è anche for-

nita una funzione che fa un set ed un check del segnale.
Questa classe è stata implementata perchè questi segnali sono frequentemente usati (c'è il rischio che si possano commettere errori se devo definire ogni volta tre variabili e sei metodi) e rendono facile la lettura di una classe.

```
class base_signal extends uvm_object;
  `uvm_object_utils(base_signal)

  // segnale predetto
  protected bit signal = 0;

  // segnale che informa se la predizione cambia
  protected bit signal_chng_expd = 0;

  // flag che dice se si deve fare un controllo o no
  protected uvm_check_e signal_check = UVM_CHECK;

  function new (string name = "base_signal");
    super.new(name);
  endfunction

  extern function void set_signal(input bit value, input uvm_check_e check = UVM_CHECK);

  extern function void set_signal_check(input uvm_check_e check = UVM_CHECK);

  extern function bit get_signal_chng_expd(input bit check_clr = 0);

  extern function void chk_signal(input bit en_ldo_h_o_v, bit wait_strup_end, bit disable_checkers);

  extern function void chk_signal_chng_expd(input bit change_flag, bit disable_checkers);

  extern task set_signal_and_check(input int unsigned wait_time = 0, bit val = 1, bit disable_checkers);

endclass
```

4.4 Lo stato del dispositivo

Lo stato del dispositivo è un'altro oggetto molto importante che deve essere gestito, vedremo nel prossimo capitolo come viene gestita la macchina a stati finiti.
Questo oggetto viene utilizzato per la gestione degli eventi di cambio di stato e contiene al suo interno un evento che verrà "triggherato" quando si vuole cambiare stato e al suo interno sono presenti anche delle variabili che danno delle informazioni riguardo allo stato corrente e quello successivo.

```
class dev_status extends uvm_object;
  uvm_event m_devstat_e;
  uvm_event_pool my_event_pool;

  //stringa che rappresenta l'enumerazione
  string curr_state_s;
  string next_state_s;

  //intero che di solito viene convertito in enumerazione
  int curr_state;
  int next_state;

  int fault_type;//intero che rappresentauna enumerazione relativa al tipo di fault

  static bit[1:0]       fault_cnt[int];//numero di fault relativi ad ogni errore
  static string flt_type[int];
  valid_t        fault_occur;//dice se è un fault valido
  `uvm_object_utils(dev_status)

  function new (string name = "dev_status");
    super.new(name);
    my_event_pool = uvm_event_pool::get_global_pool();
```

```
23      m_devstat_e = my_event_pool.get("devstat_e");
24      curr_state = 0;
25      next_state = 0;
26    endfunction
27
28    //restituisce quanti fault sono accaduti per un preciso errore
29    static function bit[1:0] get_flt(int flt);
30
31    //reset del numero di errori di un fault
32    static function void reset_flt(int flt);
33
34    //incrementa gli errori di un fault
35    static function void incr_flt(int flt);
36
37    //inizializza i fault a zero errori
38    static function void init_status(int size);
39
40    //è possibile fornire una stringa che identifica il fault
41    static function void init_flt_type(string flt_type_array[int]);
42  endclass
```

Come si può vedere dalla classe, i fault non sono noti a priori ma quando viene lanciato un evento si passa *fault_type* che è un intero che dice con che fault abbiamo a che fare, solitamente questo intero è associato ad una variabile di tipo enumerazione.
Molto spesso quando si gestiscono i fault viene anche fornito un limite di quante volte questo si può verificare prima di andare, per esempio, in un altro stato come il *FAILSAFE*.
Per questo viene fornita la variabile *fault_cnt* che associa ad ogni fault quante volte è stato lanciato e dà la possibilità di incrementarlo e/o resettarlo.
E' necessario, prima di essere utilizzato, inizializzare la variabile *fault_cnt* tramite l'appropriata funzione(*init_status*), altrimenti quando si cerca di incrementare un elemento ci sarà un errore.
Viene fornita anche un'alternativa al valore intero, dando la possibilità di attribuire una stringa allo stato(*curr_state_s*,*next_state_s*), ed è anche possibile associare ad ogni intero, che identifica uno stato, una stringa passando un array tramite la funzione *init_fault_type*.
Fare attenzione che viene dato per scontato che il valore intero degli stati identifica univocamente la sua stringa all'interno dell'array e che se si sbaglia ad assegnarne il valore con la stringa potrebbero esserci dei malfunzionamenti.

4.5 La gestione dei fault

In questa sezione approfondiamo come vengono gestiti i fault a livello di sistema e quali sono le funzioni della classe che se ne occupano.
Solitamente un ambiente di test viene organizzato in blocchi e alcuni di questi rappresentano un circuito che estende la classe base dei modelli, per esempio il Buck converter.
La gestione degli eventi generalmente viene fatta ad un livello superiore (top level) nel quale vengono chiamati i vari metodi dei modelli, che variano per ognuno di essi, e contegono le procedure da seguire.
All'interno della classe viene fornito un segnale di tipo *base_signal* che può essere utilizzato, ovviamente è possibile crearne altri in caso di bisogno.

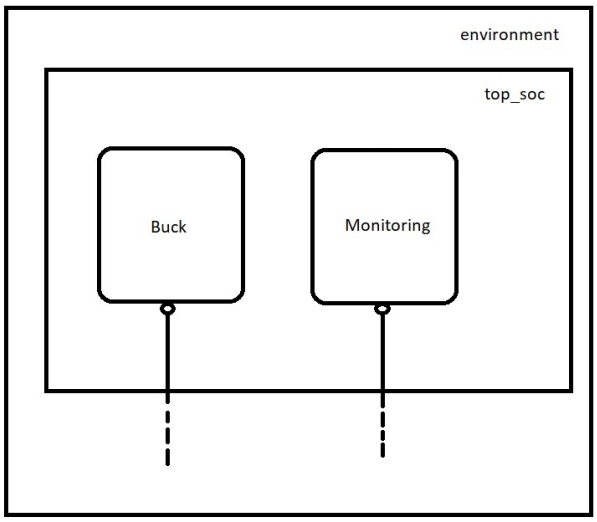

Figura 4.1: Connessione modelli

E' anche definita la funzione di write che spedisce un oggetto ,definito prima nella classe *model_item*, tramite la porta agli elementi che sono connessi.
Quando viene estesa viene passato come parametro opzionale il tipo *data_type*, che è il tipo specifico degli oggetti che verrano mandati attraverso la porta, se non viene definito è passato un valore numerico di quattro bit.

```
virtual class ifx_tb_model_base #(type data_type=bit [3:0]) extends uvm_component;

  `uvm_component_utils(ifx_tb_model_base)

  uvm_analysis_port #(model_item#(data_type)) model_port;

  protected bit disable_checkers = 0;
  protected bit disable_coverage = 0;

  //segnale per fare un enable dell'entità rappresentata
  base_signal enable_entity;
  //...
endclass
```

4.5.1 Le funzioni di gestione dei fault

Vediamo adesso più in dettaglio come vengono gestiti i fault e, come accennato in precedenza, bisogna precisare che le funzioni di gestione dei fault sono chiamate ad un livello più alto.

```
extern virtual function void set_signal_oc(input bit value, event_kind_t kind = SURE);

extern virtual function void set_signal_stg(input bit value, event_kind_t kind = SURE);

extern virtual function void set_signal_uv(input bit value, event_kind_t kind = SURE);

extern virtual function void set_signal_ov(input bit value, event_kind_t kind = SURE);

extern virtual function void set_signal_tpw(input bit value, event_kind_t kind = SURE);

extern virtual function void set_signal_tsd(input bit value, event_kind_t kind = SURE);

```

```
13 extern task automatic check_stg_rise(input bit is_set=1);
14
15 extern task automatic check_uv_rise(input bit is_set=1);
16
17 extern task automatic check_ov_rise(input bit is_set=1);
18
19 extern task automatic check_tpw_rise();
20
21 extern task automatic check_tsd_rise();
22
23 extern task automatic check_oc_rise();
```

Queste sono le funzioni che gestiscono i fault e al loro interno contengono la chiamata ad una macro che gestisce tutte le procedure e chiamate a funzioni che vanno implementate nel modello.

4.5.2 Collegamento con gli eventi

Vediamo innanzitutto che le variabili più importanti che gestiscono i fault sono degli eventi che devono essere lanciati dall'ambiente che contiene il modello tramite le funzione di esso (per esempio il Buck).

```
1  //evento Short to Ground
2  event check_stg_rise_e;
3  event check_stg_rise_sure_e;
4  event check_stg_fall_e;
5
6  //evento di undervoltage
7  event check_uv_rise_e;
8  event check_uv_rise_sure_e;
9  event check_uv_fall_e;
10
11 //evento di overvoltage
12 event check_ov_rise_e;
13 event check_ov_rise_sure_e;
14 event check_ov_fall_e;
15
16 //evento temperature warning
17 event check_tpw_rise_e;
18 event check_tpw_rise_sure_e;
19 event check_tpw_fall_e;
20
21 //evento temperature shutdown
22 event check_tsd_fall_e;
23 event check_tsd_rise_e;
24 event check_tsd_rise_sure_e;
25
26 //evento overcurrent
27 event check_oc_rise_e;
28 event check_oc_rise_sure_e;
29 event check_oc_fall_e;
```

Prendiamo come esempio una coppia di funzioni di check e set, quelle di undervoltage, e analizziamo come sono collegate fra loro e soprattutto se ci sono delle funzioni e/o variabili che devono essere definite per il corretto funzionamento.

4.5.3 Le macro di gestione

Partiamo dalla funzione di set, nel caso specifico di un fault di undervoltage. Per semplicità di lettura sono state eliminate tutte le altre varianti(overvoltage e short-to-

ground per il voltage, ci sono altre funzioni per la corrente e la temperatura).

```
`define SET_SUPPLY_ERROR(TYPE, VALUE, KIND) \
     //...
     if (KIND == POSSIBLE) begin \
          //...
          -> check_``TYPE``_rise_e; \
     end \
     else if (KIND == SURE) begin \
          //...
          -> check_``TYPE``_rise_sure_e; \
     end \
     else begin \
          //...
          -> check_``TYPE``_fall_e; \
     end
```

Quando questa funzione viene richiamata in base al tipo di evento che viene generato (rise, rise_sure e fall) da qualche parte all'interno dell'ambiente deve essere eseguita una funzione.
Questa viene eseguita nella funzione di check e si tratta di una funzione di "monitoring" che rimane sempre in ascolto che questo evento venga lanciato, eseguendo delle ruotine.

```
`define CHECK_SUPPLY_ERROR(EN_CHECK, TYPE) \
  forever begin \
    //...
    begin \
         //...
         @(check_``TYPE``_rise_e); \
         `uvm_info(get_name(),"******** POSSIBLE ev uv: disable checks ********",UVM_LOW) \
         set_uv_status_regs(CONF_CHECK, UVM_NO_CHECK); \
         uv_potent_flt_react(); \
    end \
    begin \
         //...
         @(check_``TYPE``_fall_e); \
         `uvm_info(get_name(),"******** FALL ev uv: alignment and enable checks ********",UVM_LOW) \
         set_uv_status_regs(ALIGN2CURR, UVM_CHECK); \
         uv_sp_flt_react(); \
    end \
    begin \
        //...
        @(check_``TYPE``_rise_sure_e); \
        set_uv_status_regs(PREDICT, UVM_CHECK); \
        uv_sp_flt_react(); \
    end \
    //...
  end
```

Analizziamo in dettaglio questa macro: intanto certe volte è necessario disabilitare il check. Questo serve perchè in certe occasioni non è possibile avere un valore definito per un segnale e potrebbe essere sia basso che altro(zone grige), questo viene fatto con la variabile *EN_CHECK*.
Vediamo come quando viene lanciato un evento dalla funzione set, se è stata eseguita la funzione check, questa rimane sempre in ascolto per gli eventi ed esegue delle procedure quando vengono catturati.
Ci sono due funzioni in particolare:*set_uv_status_regs* e *uv_sp_flt_react*. Ci sono funzioni simili anche per tutti gli altri fault e in generale queste funzioni devono essere implementate quando viene estesa la classe perchè ogni modello ha dei registri particolari che deve settare e reagisce in maniera diversa ad un fault.
La prima funzione, *set_uv_status_regs*, come si può intuire, setta dei registri, mentre la

seconda, *uv_sp_flt_react*, ha una serie di procedure che variano in base al fault e in genere potrebbero esserci quattro alternative: avere solamente un warning e rimanere nello stesso stato, andare in *FAILSAFE*, andare nello stato di *POWERDOWN* oppure fare una transizione in *INIT*.
Come verrà spiegato nel capitolo successivo, gli stati *POWERDOWN*, *INIT* e *FAILSAFE* sono considerati come stati necessari e infatti sono definite delle funzione che innescano un cambio di stato verso questi.

```
1 extern virtual function void sev_flt_react(int flt,bit mask, valid_t occurrence);
2 extern virtual function void flt2init_react(int flt,bit mask, valid_t occurrence);
3 extern virtual function void flt2powerdown_react(int flt,bit mask, valid_t occurrence);
4 extern virtual function void warn_flt_react(int flt,bit mask, valid_t occurrence);
```

5

La macchina a stati finiti

5.1 La classe base di uno stato

La macchina a stati finiti è una parte molto importante di un circuito integrato digitale, perchè in generale sono definiti degli stati nei quali ci sono delle operazioni che è necessario fare quando si entra e/o quando si esce da uno stato.

5.1.1 I metodi do_state e check_state

La fase di entrata in uno stato viene chiamata funzione *do_state()* che esegue una serie di operazioni da svolgere. Al contrario, la fase di uscita da uno stato viene chiamata

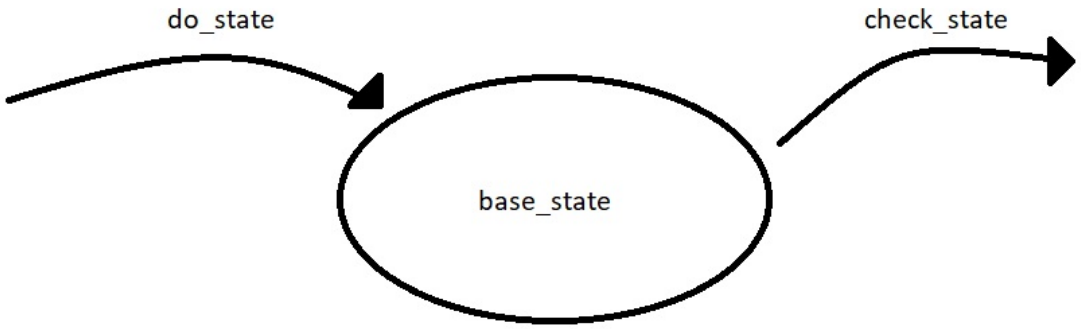

Figura 5.1: Transizione di stati

funzione *check_state()* che esegue le operazioni di chiusura.

5.1.2 Le variabili di stato

Nella libreria viene fornita una classe di base virtualmente vuota con qualche variabile di stato importante per la gestione.

```
static true_false_t sevflt_occ = FALSE; // per identificare se un fault severe è accaduto
static true_false_t flt2init_occ = FALSE; // dice se un fault che fa una transiziona a INIT è accaduto
static true_false_t warnflt_occ = FALSE;  // dice se un fault che fa rimanere nello stesso stato è accaduto

uvm_event m_devstat_e;
dev_status m_dev_status;//stato del dispositivo
int id_state;
string m_state_name = "NONE";//nome dello stato
string m_next_state_name = "NONE";//prossimo stato
string m_valid_states[$];//transizioni di stati valide
```

Ovviamente è necessario estendere questa classe per ogni stato che si pensa di avere nella FSM, ridefinendo il metodo *do_state()* e *check_state()* perchè, come si vedrà nelle sezioni successive, verranno utilizzati automaticamente quando ci sarà un salto di stato.

5.2 La classe Flow State

In questa classe viene inizializzata la macchina a stati finiti e si ha a che fare non con il singolo stato, ma con un array di stati, a cui è associato un id, che vengono passati e caricati.
Ovviamente quando definisco uno stato devo non solo crearlo e definire le funzioni base di do e check, ma devo anche attribuirgli un nome ed un id che li identifica.

5.2.1 Le variabili

Queste sono le variabili della classe e sono comuni: alcune si riferiscono ad uno stato preciso, altre invece sono comuni a tutto l'ambiente che racchiude gli stati.

```
static string m_flow[string][$]; //stati possibili
static ifx_fsm_state m_cur_fsm_state;//stato corrente
static ifx_fsm_state m_prev_fsm_state;//stato precedente
static string m_cur_fsm_state_name = "NONE";//settato allo stato zero all'inizio

protected static int initial_state_id='INIT_STATE_ID;//state_id=1 di default
protected static int powerdown_state_id='POWERDOWN_STATE_ID;//state_id=0 di default
protected static int failsafe_state_id='FAILSAFE_STATE_ID;//state_id=2 di default
protected static true_false_t is_init_set=FALSE;
protected static true_false_t is_powerdown_set=FALSE;
protected static true_false_t is_failsafe_set=FALSE;
//queste due variabili devono essere settate: prima fsm_list tramite set_fsm_flow_list, poi m_fsm_list
//chiamando init_fsm_flow
static ifx_fsm_state m_fsm_list[string];//lista di possibili stati che estendono ifx_fsm_state accessibili tramite fsm_list
static string fsm_list [];//lista dei nomi dei possibili stati
```

Per esempio ci sono delle variabili di tipo *true_false_t*(zero o uno logico) perchè ci sono delle funzioni che permettono di settare le tre variabili associate (*failsafe_state_id*, *powerdown_state_id* e *initial_state_id*) e queste variabili sono settabili una sola volta, quindi questi valori logici servono a memorizzare se il valore è già stato cambiato.
Ovviamente si sconsiglia di settarlo dopo la fase di inizializzazione perchè questo può

causare errori visto che questi valori sono utilizzati dalla classe dei modelli quando c'è una reazione a dei fault.

5.2.2 Le funzioni

Queste sono le funzioni di base fornite nella classe fsm_flow e permettono di creare un ambiente per la macchina a stati finiti.

```
1  static function void init_fsm_flow(uvm_component fsm_scb = null, ifx_fsm_state state_collection []);
2
3  //dobbiamo noi passare gli stati possibili
4  static function void set_fsm_flow_list(string fsm_list_flow []);
5
6  //supponiamo che lo stato zero sia quello iniziale
7  static function ifx_fsm_state get_initial_state();
```

Bisogna stare attenti perchè nella macchina a stati finiti è dato per scontato che la prima cella dell'array che contiene gli stati è anche considerata come lo stato di partenza.
Le tre funzioni permettono, in ordine, di inizializzare la macchina a stati finiti passando un array degli stati contenuti al suo interno, di passare una lista di stringhe che identifica tutti gli stati e di ritornare lo stato iniziale.

5.3 La classe di gestione

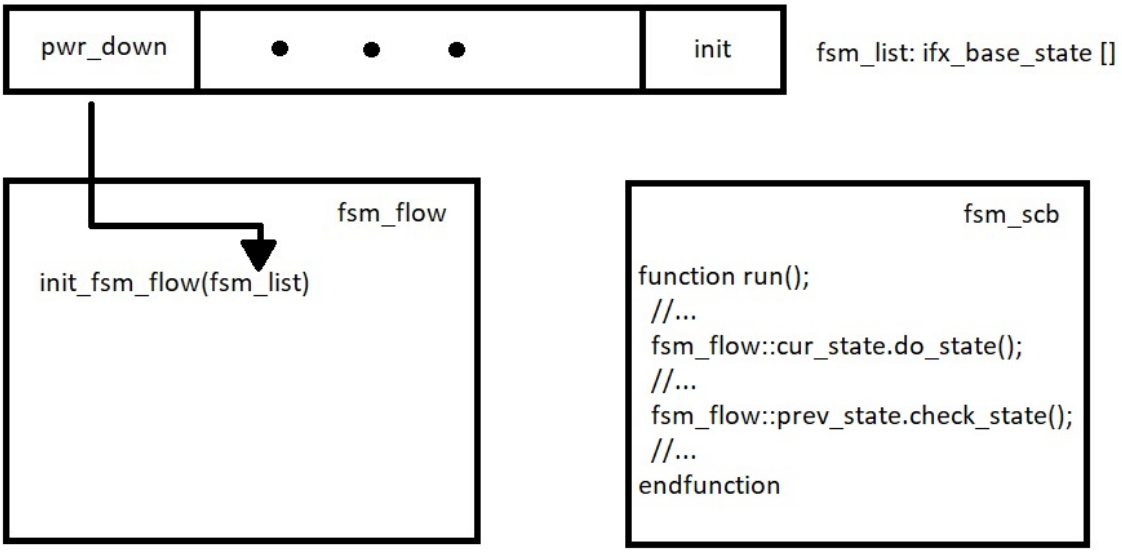

Figura 5.2: Gestione degli stati

```
task run_phase(uvm_phase phase);
    check_state_ids();
    set_cur_fsm_inst();
    forever begin
          //...
        fsm_flow::m_cur_fsm_state.do_state();
      //...
      fsm_flow::m_prev_fsm_state.check_state(fsm_flow::m_cur_fsm_state.m_next_state_name);
      //...
    end
    set_cur_fsm_inst();
end
endtask
```

Come si può vedere quello che questa classe fa è molto semplice, parte dallo stato iniziale e rimane in ascolto se ci sono dei cambi di stato ed esegue il *check_state()* uscendo e poi entrando nel nuovo stato esegue il *do_state()*.

5.3.1 Gli stati necessari

Leggendo attentamente il codice sopra si può notare che esiste una funzione che si chiama *check_state_ids()* che esegue un controllo.
Questo è stato implementato perchè viene dato per scontato che esistano tre stati: Powerdown, Init e Failsafe. In questa routine si scorre l'array che contiene gli stati e viene confrontato l'id che è presente nei singoli stati con quello definito nella classe *flow_state* per i tre stati base, ovviamente se c'è una discordanza la simulazione si ferma con un messaggio di errore.
Per precisare, il controllo verifica la sola presenza di uno stato con lo stesso id, questo

non vuol dire necessariamente che quello stato sia uno di quello base.
Sta infatti all'utilizzatore accertarsi di questa cosa poichè non è possibile identificare univocamente gli stati base, visto che viene passato un array che contiene il gruppo e non quelli singoli.

5.3.2 Esempio di utilizzo

Allego qui un esempio di creazione di una macchina a stati finiti contenente tre stati: *powerdown*, *init* e *failsafe*.
Viene creata una sequenza che induce delle transizioni dallo stato di *POWERDOWN*, quello iniziale, a *INIT*, di nuovo in *POWERDOWN* e poi *FAILSAFE*, infine si prova ad andare in *WAKE*, ma questo stato non esiste e crea un errore.

```
//Definisco gli stati possibili

class ifx_fsm_init_ex extends ifx_fsm_state ;

  static local ifx_fsm_init_ex m_inst = null;

  `uvm_component_utils(ifx_fsm_init_ex)

  function new(string name = "fsm_state", uvm_component parent = null);
    super.new(name, parent);
  m_state_name="INIT";
  state_id=1;
  endfunction

  virtual task do_state();
    `uvm_info(get_name(),"INIT state",UVM_LOW)
    forever begin
        m_devstat_e.wait_trigger();
        $cast(m_dev_status, m_devstat_e.get_trigger_data());
      case(m_dev_status.next_state)
        ST_INIT:begin
            `uvm_info(get_name(),"Already in INIT state",UVM_LOW)
            m_next_state_name = "INIT";
        end
        ST_FAILSAFE:begin
            `uvm_info(get_name(),"Moving to FAILSAFE state",UVM_LOW)
            m_next_state_name = "FAILSAFE";
          break;
        end
        ST_POWERDOWN:begin
            `uvm_info(get_name(),"Moving to POWERDOWN state",UVM_LOW)
            m_next_state_name = "POWERDOWN";
          break;
        end
        default:`uvm_error(get_name(),"state not valid")
      endcase
    end
  endtask

  virtual task check_state(string m_next_state_name);
    `uvm_info(get_name(),"Check INIT state",UVM_LOW)
  endtask
endclass

class ifx_fsm_powerdown_ex extends ifx_fsm_state;

  static local ifx_fsm_powerdown_ex m_inst = null;

  `uvm_component_utils(ifx_fsm_powerdown_ex)

  function new(string name = "fsm_state", uvm_component parent = null);
    super.new(name, parent);
  m_state_name="POWERDOWN";
  state_id=0;
  endfunction

  virtual task do_state();
    `uvm_info(get_name(),"POWERDOWN state",UVM_LOW)
    forever begin
        m_devstat_e.wait_trigger();
```

```
61          $cast(m_dev_status, m_devstat_e.get_trigger_data());
62        case(m_dev_status.next_state)
63          ST_INIT:begin
64              `uvm_info(get_name(),"Moving to INIT state",UVM_LOW)
65              m_next_state_name = "INIT";
66            break;
67          end
68          ST_FAILSAFE:begin
69              `uvm_info(get_name(),"Moving to FAILSAFE state",UVM_LOW)
70              m_next_state_name = "FAILSAFE";
71            break;
72          end
73          ST_POWERDOWN:begin
74              `uvm_info(get_name(),"Already in POWERDOWN state",UVM_LOW)
75              m_next_state_name = "POWERDOWN";
76          end
77          default:`uvm_error(get_name(),"state not valid")
78        endcase
79      end
80    endtask
81
82    virtual task check_state(string m_next_state_name);
83      `uvm_info(get_name(),"Check POWERDOWN state",UVM_LOW)
84    endtask
85
86  endclass
87
88  class ifx_fsm_failsafe_ex extends ifx_fsm_state;
89
90    static local ifx_fsm_failsafe_ex m_inst = null;
91
92    `uvm_component_utils(ifx_fsm_failsafe_ex)
93
94    function new(string name = "fsm_state", uvm_component parent = null);
95      super.new(name, parent);
96      state_id=2;
97    m_state_name="FAILSAFE";
98    endfunction
99
100   virtual task do_state();
101     `uvm_info(get_name(),"FAILSAFE state",UVM_LOW)
102     forever begin
103         m_devstat_e.wait_trigger();
104         $cast(m_dev_status, m_devstat_e.get_trigger_data());
105       case(m_dev_status.next_state)
106         ST_INIT:begin
107             `uvm_info(get_name(),"Moving to INIT state",UVM_LOW)
108             m_next_state_name = "INIT";
109           break;
110         end
111         ST_FAILSAFE:begin
112             `uvm_info(get_name(),"Already in FAILSAFE state",UVM_LOW)
113             m_next_state_name = "FAILSAFE";
114
115         end
116         ST_POWERDOWN:begin
117             `uvm_info(get_name(),"Moving to POWERDOWN state",UVM_LOW)
118             m_next_state_name = "POWERDOWN";
119           break;
120         end
121         default:`uvm_error(get_name(),"state not valid")
122       endcase
123     end
124   endtask
125
126   virtual task check_state(string m_next_state_name);
127     `uvm_info(get_name(),"Check FAILSAFE state",UVM_LOW)
128   endtask
129 endclass
130
131 class fsm_scb_seq_ex extends uvm_sequence #(uvm_sequence_item);
132
133   `uvm_object_utils(fsm_scb_seq_ex)
134   dev_status m_dev_status;
135     function new(string name = "fsm_scb_seq");
136       super.new(name);
137     m_dev_status=new();
138     endfunction: new
139     //m_devstat_e è un enumerazione con al suo interno degli stati {ST_POWERDOWN,ST_INIT,ST_FAILSAFE,ST_WAKE} e per ques'
        ultimo non è definito uno stato
140     task body();
141       //Si innesca un cambio di stati da ST_POWERDOWN(lo stato iniziale) a ST_INIT, per poi passare a ST_FAILSAFE e infine si
        prova ad andare in ST_WAKE ma c'è un errore perchè questo stato non è definito
```

```
142     m_dev_status.next_state=ST_INIT;
143       m_dev_status.m_devstat_e.trigger(m_dev_status);
144     #5ns;
145     m_dev_status.next_state=ST_POWERDOWN;
146       m_dev_status.m_devstat_e.trigger(m_dev_status);
147     #5ns;
148     m_dev_status.next_state=ST_FAILSAFE;
149       m_dev_status.m_devstat_e.trigger(m_dev_status);
150     #5ns;
151     m_dev_status.next_state=ST_WAKE;
152       m_dev_status.m_devstat_e.trigger(m_dev_status);
153       endtask: body
154 endclass
155
156
157 class fsm_state_example extends uvm_test;
158
159   `uvm_component_utils(fsm_state_example)
160
161   fsm_scb_seq_ex seq;
162
163   function new(string name = "hysop_faults_test_test", uvm_component parent);
164     super.new(name, parent);
165   endfunction: new
166       virtual function void build_phase(uvm_phase phase);
167     ifx_fsm_init_ex init=new("init",this);
168     ifx_fsm_failsafe_ex failsafe=new("failsafe",this);
169     ifx_fsm_normal_ex powerdown=new("powerdown",this);
170     fsm_scb m_fsm_scb;
171     dev_status m_dev_status;
172       super.build_phase(phase);
173     m_dev_status=new();
174     fsm_flow::set_fsm_flow_list({"POWERDOWN","INIT","FAILSAFE"});
175     fsm_flow::init_fsm_flow(this,{powerdown,init,failsafe});
176     m_fsm_scb = fsm_scb::type_id::create("m_fsm_scb", this);
177     endfunction: build_phase
178
179   task run_phase(uvm_phase phase);
180     super.run_phase(phase);
181     seq = fsm_scb_seq_ex::type_id::create("fsm_state_example", this);
182     phase.raise_objection(this);
183     seq.start(null);
184     phase.drop_objection(this);
185   endtask : run_phase
186
187 endclass: fsm_state_example
```

6

Un ambiente per monitorare segnali

6.1 Introduzione

Una delle funzionalità più importanti di un ambiente di verifica è il monitor, questo perchè semplifica molto il debug monitorando i segnali chiave e le cause del loro malfunzionamento.
Ovviamente non è possibile sapere perchè funzionano in modo diverso da quello che si pensava e per fare questo è necessario analizzare il test nel suo insieme però sapendo quale segnale non si comporta correttamente si può restringere il campo e identificare più facilmente la causa del problema.

6.2 Trasmettere dati:la classe base item

Come per la classe dei modelli anche il monitor ha un suo oggetto che deve essere trasferito tramite una porta ad un altro modulo, solitamente viene collegato allo scoreboard.

```
1  class ifx_base_seq_item #(type base_source_type=bit [23:0],type base_data_type=bit [23:0]) extends uvm_sequence_item;
2
3    `uvm_object_utils(ifx_base_seq_item)
4
5    rand ifx_cmd_t    cmd;     // comando [VOLTAGE,CURRENT,...]
6    rand base_source_type source; // identifica la provenienza
7    rand base_data_type data; //il dato inviato, di solito una enumerazione
8    rand event_kind_t kind; // kind : glitch, possible, sure
9    rand bit wait_for_enable;
10   rand bit wait_rsp;
11   rand int period;
12   rand int frequency;
13
14   int unsigned id;//identifica l'oggetto
15   static int unsigned s_id; //crea un id unico
16
17   extern function void new(string name="item");
18   extern function string convert2string();
19   extern function ifx_base_seq_item clone(ifx_base_seq_item item);
20
21 endclass
```

Come si può vedere quando definisco questo oggetto devo passare due tipi di dati: il *source_data_type* che corrisponde alla provenienza del dato e il *base_data_type* che invece è effettivamente il valore che voglio trasmettere. Entrambi solitamente sono enumerazioni.

6.3 Il Driver

Come visto nel capitolo due, il monitor è quel blocco che fa da tramite fra la sequenza che voglio introdurre nell'ambiente e il dispositivo che voglio testare, andando a modificare i segnali dell'interfaccia.

```
1 extern virtual protected task drive();
2 extern virtual protected task get_item(bit enable_mbx);
3 extern virtual protected task drive_voltage_ldo_qvr(event_kind_t kind, voltage_t volt, bit wait_for_enable);
4 extern virtual protected task drive_temperature(temperature_t temperature, event_kind_t kind, bit wait_for_enable);
5 extern virtual protected task drive_current(current_t current, event_kind_t kind, bit wait_for_enable);
6 extern virtual task case_drive(event_kind_t kind, ifx_cmd_t type_cmd, bit wait_for_enable);
```

Queste sono le funzioni di base fornite nel driver e adesso analizziamo come sono connesse fra loro.
Intanto cominciamo da *get_item()*, questa funzione deve essere chiamata inizialmente e rimane in ascolto per delle sequenze.
Come è stato spiegato nel capitolo due, quando viene dato il via ad una sequenza, quest'ultima è mandata dal sequencer al driver tramite una porta. Questo metodo permette di mettere questo oggetto in una *mailbox*.
Adesso entra in gioco la funzione *drive()*, anche questa in ascolto, però qui si guarda la *mailbox* che mano a mano che viene riempita verrà svuotata e verrà fatto un "driving" dei segnali.
Quando viene ricevuto un elemento dalla mailbox viene chiamata la funzione *case_drive()* che semplicemente "sceglie" che tipo di segnale deve pilotare: corrente,tensione o temperatura.
Questi sono i segnali di base poi se è necessario averne altri la classe può essere estesa e modificata.
A questo proposito bisogna specificare che il driving effettivo dei segnali non è fatto e ci sono tre funzioni(per temperatura, corrente e tensione) che contengono solamente una chiamata ad un UVM error che devono essere implementate.
Questo perchè non è possibile sapere a priori che segnali devono essere settati, di solito ogni circuito ha le sue procedure per il pilotaggio dei segnali che sono decise in fase di progetto e soprattutto è necessario avere a disposizione un'interfaccia del dispositivo che in questa fase non si ha.
Sono anche fornite delle funzioni che fanno una scelta di che segnale si vuole mandare all'interfaccia e che si possono utilizzare quando viene implementato il pilotaggio del

segnale.

```
extern virtual task automatic case_current(current_t current, event_kind_t kind, bit wait_for_enable,ref int curr_duration);
extern virtual task automatic case_temp(temperature_t temperature, event_kind_t kind, bit wait_for_enable, ref int
    temp_duration);
extern virtual task automatic case_volt(voltage_t volt, event_kind_t kind, bit wait_for_enable, ref int volt_duration);
```

Queste funzioni semplicemente ricevono un evento e randomizzano un segnale scegliendo, dato il valore di kind, che tipo di evento pilotare (*GLITCH*, *POSSIBLE* o *SURE*) e decidono anche il tempo con dei valori all'interno dei parametri consentiti, ci sono delle variabili che variano per i tre casi di tensione,corrente e temperatura e queste variabili hanno un valore minimo ed uno massimo in cui possono assumere i valori.

```
extern function void set_rand_voltage(voltage_t volt, time duration = 0);
extern function void set_monitor_curr(current_t current, time duration = 0);
extern function void set_monitor_temp(temperature_t temperature, time duration = 0);
```

Queste funzioni sono chiamate all'interno del driver quando bisogna scegliere il tipo di evento da chiamare e devono essere collegate all'interfaccia, specificatamente nell'interfaccia verrà chiamato un evento che farà partire una procedura.

6.4 Il file di configurazione

In questo file vengono messe tutte le variabili che sono necessarie a livello del monitor, principalmente sono segnali temporali che, come vedremo, hanno valori diversi in base a quale modulo li ha chiamati e serviranno nelle ruotine di monitoring dei segnali.

```
//Tutti questi sono array perchè si possono avere tempi diversi in base alla provenienza base_source_type
//per la tensione
int stg_delay_min [];
int stg_delay_max [];
int uv_deglitch_min [];
int uv_deglitch_max [];
int ov_deglitch_min [];
int ov_deglitch_max [];

//per la corrente
int t_ov_stby_deglitch_min [];
int t_ov_stby_deglitch_max [];
int t_uv_stby_deglitch_min [];
int t_uv_stby_deglitch_max [];

//per la temperatura
int t_rr_dig_min;
int t_rr_dig_max;
```

6.5 Il monitor

In questa sezione analizziamo in dettaglio tutte le funzioni che vengono fornite per monitorare segnali di vario tipo.

6.5.1 La coverage

Incominciamo facendo una introduzione alla coverage che è una parte molto importante e fornisce un resoconto di quello che è successo nelle simulazioni, dalle variabili che sono state randomizzate fino agli errori che ne sono conseguiti dando una visione di insieme sui casi che sono emersi e quelli che mancano da simulare.

```
task automatic ifx_base_monitor::ana_monitor_cover();

protected event cov_transaction; // event to trigger coverage

covergroup cov_trans @cov_transaction;
  option.per_instance = 1;
  cp_cmd: coverpoint cover_item.cmd;
  cp_source: coverpoint cover_item.source{
    }
  cp_voltage: coverpoint voltage_t'(cover_item.data);
  cp_temperature: coverpoint temperature_t'(cover_item.data);
  cp_current: coverpoint current_t'(cover_item.data);
  cp_kind: coverpoint cover_item.kind;

  //Voltage monitor coverage
  x_voltage_source_kind: cross cp_cmd, cp_source, cp_voltage, cp_kind {
    ignore_bins ignored =
      !binsof (cp_cmd) intersect {VOLT}|| !binsof (cp_kind) intersect {GLITCH, SURE, SET_VALUE}
      || !binsof (cp_voltage) intersect {UV, OV, STG};
  }

  //Temperature monitor coverage
  x_temperature_source_kind: cross cp_cmd, cp_source, cp_temperature, cp_kind {
    ignore_bins ignored =
      !binsof (cp_cmd) intersect {TEMP}
      || !binsof (cp_kind) intersect {GLITCH, SURE, SET_VALUE}
      || !binsof (cp_temperature) intersect {TSD};
  }

  //Current monitor coverage
  x_current_source_kind: cross cp_cmd, cp_source, cp_current, cp_kind {
    ignore_bins ignored =
      !binsof (cp_cmd) intersect {CURRENT}
      || !binsof (cp_kind) intersect {GLITCH, SURE, SET_VALUE}
      || !binsof (cp_current) intersect {OC,UC};
  }

endgroup: cov_trans
```

Questo è un *covergroup* molto generale che è stato inserito all'interno del monitor e viene lanciato quando l'evento *cov_transaction* viene triggherato e semplicemente salva il punto operativo aggiungendolo alla "collezione" per essere poi esaminato aiutando nell'individuare la direzione da prendere per test successivi.
La funzione *ana_monitor_cover* è sempre in ascolto e controlla che una *mailbox*, *cov_mailbox*, venga riempita e quando questa ha degli elementi, che vengono aggiunti dal monitor, lancia l'evento che fa il sampling del *covergroup*.

6.5.2 Monitorare un segnale di tensione,corrente e temperatura

Il monitoraggio di un segnale di tensione, corrente e temperatura hanno delle routine di gestione praticamente identiche.
Per evitare di essere ridondante viene analizzato solo il monitoraggio della tensione ma il comportamento degli altri due e le variabili in gioco sono simili.
Il monitoraggio di una tensione ha più di una funzionalità andando a monitorare non solo se ci sono degli sbalzi di tensione ma anche il tipo di impulso che è stato rilevato.

```
1  extern protected task monitor_voltage(base_source_type source, ref logic vif_uv, ref logic vif_ov, ref logic vif_enable =
       dummy_zero, input realtime enable_req_time = 0,bit enable_abist=1);
2
3  extern protected task deglitch_vcond(voltage_t vcond, base_source_type source, input realtime en_time = 0, input realtime
       en_req_time = 0);
4
5  extern protected task handle_vcond(voltage_t vcond, base_source_type source, output process p, input realtime en_time = 0,
       input realtime en_req_time = 0);
6
7  extern protected task kill_deglitch_filter(base_source_type source, voltage_t cause, process p);
8
9  extern protected task report_source_voltage(base_source_type source, bit is_stuck_flt = 0);
10
11 extern protected task report_source_coverage(base_source_type source, ifx_cmd_t cmd=CURR, bit[23:0] data=OC, event_kind_t
       kind=SET_VALUE);
```

La prima funzione che viene chiamata è *monitor_voltage*, è necessario passare il pad che monitora l'overvoltage e l'undervoltage, l'enable non è necessaria.
La funzione controlla inizialmente che questi tre pad non abbiano dei valori non definiti e se viene passato il segnale di enable si salva il momento in cui questo segnale cambia valore per effettuare successivamente dei controlli.
Adesso è il momento di controllare se c'è stato un meno uno sbalzo di tensione(overvoltage o undervoltage) e ci sono due casi: il primo è il caso in cui ci sia un rise e in questo caso viene chiamata una funzione(*handle_vcond*) che gestisce e identifica il tipo di fault introdotto, mentre il secondo ho un evento di fall che chiama in successione tre funzioni(*kill_deglitch_filter*,*report_source_coverage* e *report_source_voltage*).
Partiamo da analizzare *handle_vcond* la quale crea un processo e chiama la funzione *deglitch_vcond*, quest'ultima al suo interno definisce delle variabili.

```
1  int stg_delay_min=t_config.stg_delay_min[source];
2  int stg_delay_max=t_config.stg_delay_max[source];
3  int uv_deglitch_min=t_config.uv_deglitch_min[source];
4  int uv_deglitch_max=t_config.uv_deglitch_max[source];
5  int ov_deglitch_min=t_config.ov_deglitch_min[source];
6  int ov_deglitch_max=t_config.ov_deglitch_max[source];
```

Queste variabili sono dei tempi da aspettare che variano in base alla provenienza di questo fault e serviranno per assegnare un valore a *timeout_potential* e *timeout_sure* in base a che tipo di sbalzo di tensione è accaduto(*OV*,*UV* o *STG*).
Successivamente viene mandato un oggetto dicendo che è successo un evento di tipo *GLITCH*, si aspetta *timeout_potential* e poi si manda un evento *POSSIBLE* e infine aspetto la differenza fra i due tempi(*timeout_sure-timeout_potential*) e mando un evento di tipo *SURE*.
In contemporanea a questo processo la funzione *monitor_voltage* sta sempre andando

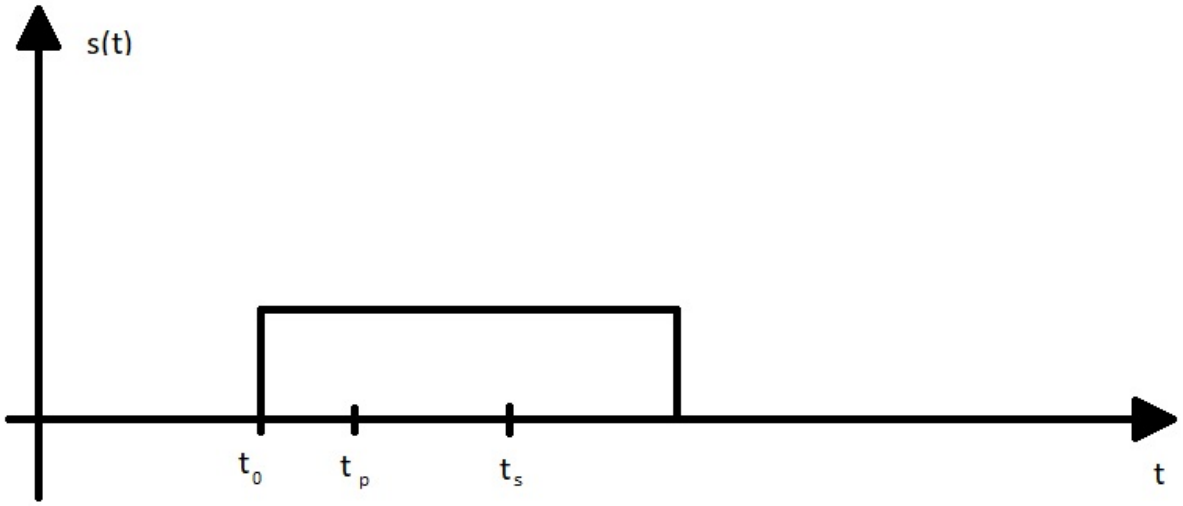

Figura 6.1: Disturbo introdotto

e se arriva un evento di fall quello che succede, come abbiamo accennato prima, è che viene chiamata la funzione *kill_deglitch_filter* che se il processo non è già terminato lo elimina, il che vuol dire che se ho inviato una notifica di un *GLITCH* e ho un evento di fall le notifiche di *POSSIBLE* e *SURE* non arriveranno, giustamente, mai.
Inoltre questa routine invia anche dei report della coverage.
Per completezza metto anche le funzione di gestione di corrente e temperatura elencando le differenze che ci sono con il monitoraggio della tensione.

```
1  //funzione di monitoring della corrente
2  extern protected task monitor_current(base_source_type source, ref logic vif_uc, ref logic vif_oc, ref logic vif_enable =
       dummy_zero, input realtime enable_req_time = 0);
3  extern protected task handle_ccond(current_t ccond, base_source_type source, output process p, input realtime en_time = 0,
       input realtime en_req_time = 0);
4  extern protected task kill_deglitch_current(base_source_type source, process p);
5  extern protected task deglitch_ccond(current_t ccond, base_source_type source, input realtime en_time = 0, input realtime
       en_req_time = 0,input bit enable_time,input bit done=0);
6  extern protected task report_current(base_source_type source,event_kind_t kind=NO_KIND, bit is_stuck_flt = 0);
7
8  //funzioni di monitoring della temperatura
9  extern protected task monitor_temperature(ref vif_enable, ref vif_tsd, input base_source_type source);
10 extern protected task deglitch_tcond(temperature_t tcond, base_source_type source);
11 extern protected task handle_tcond(temperature_t tcond, output process p, base_source_type source);
12 extern protected task kill_deglitch_temp(temperature_t tcond, process p, base_source_type source);
13 extern protected task report_temp(temperature_t tcond, base_source_type source);
```

Le variabili che gestiscono i tempi delle correnti e delle temperature sono le seguenti.

```
1  //deglitch della corrente
2  if (ccond == OC) begin
3      timeout_potential = t_config.t_ov_stby_deglitch_min[source];
4      timeout_sure = t_config.t_ov_stby_deglitch_max[source];
5  end
6
7  if (ccond == UC) begin
8      timeout_potential = t_config.t_uv_stby_deglitch_min[source];
9      timeout_sure = t_config.t_uv_stby_deglitch_max[source];
10 end
11
12 //deglitch della temperatura
13 timeout_potential = t_config.t_rr_dig_min;
14 timeout_sure = t_config.t_rr_dig_max;
```

A parte le differenze per quanto riguarda i tempi e i diversi errori che possono verificarsi la gestione a livello di funzioni e di flusso e la stessa delle tensioni.

6.5.3 Monitorare un bus di N bit

In questa sezione parleremo di un monitor per un bus di N bit e, come avevamo accennato in precedenza nel capitolo due , questo è un caso particolare in cui è necessario utilizzare una macro.
E' necessaria perchè system verilog è molto stringente per quanto riguarda gli array, ovvero è necessario che quando vengono definiti gli indici di inizio e fine siano delle costanti.
Se però si vuole monitorare un segnale digitale con N bit e non si conosce a priori quanti sono, ci sono due soluzioni: creare una funzione per ogni valore di N, che risulta ridondante ed inefficiente, oppure creare una macro.
Quest'ultima dà la possibilità di togliere questa limitazione e, come si può vedere nel codice qui sotto, monitora un segnale ma non sa effettivamente che tipo di segnale è, anche se è comunque in grado di adattarsi.

```
`define MONITOR_N_BIT_BUS(MODEL,SIGNAL, SOURCE) \
  //...
  forever begin \
  //...
    @(``MODEL``.``SIGNAL``) begin\
        //...
        vif_signal_x: assert (!$isunknown(``MODEL``.``SIGNAL``)) //...
        //invia il report della coverage
        monitor_port.write(item);\
    end \
  end
```

Come si può vedere è necessario passare un segnale, visto come una stringa dalla macro, il blocco di provenienza e la variabile che contiene questo segnale, solitamente è contenuto nell'interfaccia e questa viene passata utilizzando il configuration database.

6.5.4 Monitorare un segnale raw

Questa routine serve per controllare se un segnale ha subito o meno un evento di overvoltage o di undervoltage.
Ogni volta che il segnale che controlla l'overvoltage e/o quello che controlla l'undervoltage è alto si controlla se quello precedente era alto o basso e viene inviato un dato tramite la porta che rispecchia questa condizione.

```
task automatic ifx_base_monitor::monitor_raw_voltage(ref logic vif_uv, ref logic vif_ov, input base_source_type source);
  automatic logic cur_uv = 1'b0;
  automatic logic cur_ov = 1'b0;
  logic uv_act_lvl = 1;
  //...
  forever begin
    @(vif_uv or vif_ov);
```

```
8     if (vif_ov === 1'b0 && cur_ov === 1'b1 && vif_uv === !uv_act_lvl) begin
9       // OV presente -> non present; UV non presente
10    end else if (vif_ov === 1'b1 && cur_ov === 1'b0) begin
11      // OV non presente -> presente
12    end
13    if (vif_uv === !uv_act_lvl && cur_uv === uv_act_lvl && vif_ov === 1'b0) begin
14      // UV presente -> non presente; OV non presente
15    end else if (vif_uv === uv_act_lvl && cur_uv === !uv_act_lvl) begin
16      // UV non presente -> presente
17    end
18    monitor_port.write(item);
19    cur_ov = vif_ov;
20    cur_uv = vif_uv;
21  end
22 endtask: monitor_raw_voltage
```

Notiamo che ci sono due segnali che devono essere passati, li consideriamo segnali digitali e mi dicono se c'è un overvoltage o undervoltage, schematicamente possono essere considerati come le uscite di due comparatori di tensione.

6.5.5 Monitorare un segnale di clock

Molto spesso è utile monitorare la frequenza di un segnale per vedere se sta funzionando correttamente.
La funzione qui sotto fa proprio questo e necessita del passaggio, oltre che del segnale, anche della provenienza e dei limiti (minimo e massimo) consentiti per il periodo del segnale.

```
1  task automatic ifx_base_monitor::monitor_clk_mng_raw_clk(ref logic vif_signal, input base_source_type source, int t_clk_ok_min
       , int t_clk_ok_max);
2  forever begin
3      //...
4      @(vif_signal);
5          if (vif_signal === 1'b1 && prev_time != 0) begin
6              //calcola il periodo
7            if ((period >= t_clk_ok_min) && (period <= t_clk_ok_max) ) begin
8              //range del clock corretto
9            end
10           if (period < t_clk_ok_min) begin
11             //periodo di clock basso
12           end
13           if (period > t_clk_ok_max) begin
14              //periodo di clock troppo alto
15           end
16           //...
17           //report event if period change
18         end
19       else if (vif_signal === 1'b1)
20         prev_time = $time;
21       end
22     end
23     //...
24 end
25 endtask
```

La funzione è molto semplice: la prima volta che nota un cambiamento salva il tempo (*prev_time*) e se questo segnale cambia ancora viene calcolato il periodo.
Se non rientra nei canoni voluti viene mandato un messaggio di errore.
Inoltre viene mandato un dato, solitamente allo scoreboard, per notificarlo di questo eventuale errore e nel caso effettuare delle procedure se necessario.

7 Lo scoreboard

7.1 Funzionalità

Per quanto riguarda gli scoreboard bisogna parlare prima di tutto delle sue funzionalità. Questo modulo è collegato con tutti i modelli ed il monitor e la sua funzionalità è quella di gestire tutti i pacchetti di errori/warning che gli vengono inviati tramite le porte e fare le dovute operazioni.

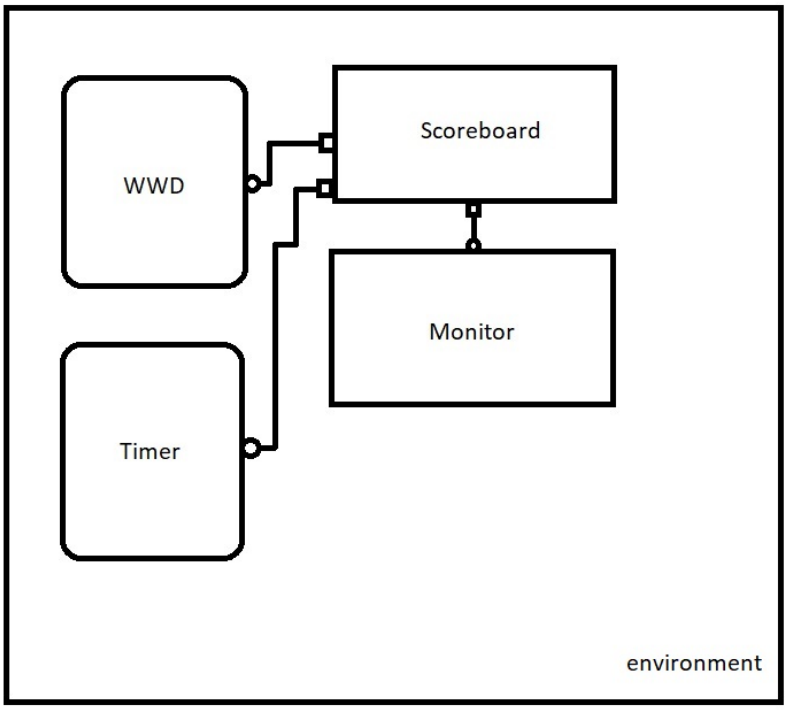

Figura 7.1: Collegamenti con lo scoreboard

Ci sono diversi modi di implementazione di questa gestione degli oggetti: si potrebbe creare una porta e gestire separatamente con un metodo di write tutti i possibili scenari di errore che possono arrivare oppure si potrebbe utilizzare la particolarità della classe vista prima (per i modelli) e raggruppare in una unica porta tutti i modelli e gestirli inizialmente in base al tipo di dato che viene inviato, e poi più specificatamente in base all'errore che vogliono andare ad indirizzare.

7.2 La gestione dei registri

Nella sezione precedente è stato analizzato in generale cosa deve fare lo scoreboard, in particolare di solito viene istanziato un ambiente, *top_soc*, che contiene tutti i modelli e che invia dati allo scoreboard, è diverso dai singoli pacchetti dei diversi modelli.
Infatti questo si occupa di gestire i valori di certi registri che devono essere gestiti e di fare la prediction e/o l'update di questi valori.

```
function int get_field_index(input string reg_name = "", input string field_name = "");

function string get_field_access(input string reg_name = "", input string field_name = "");

function void predict_field(input string rg_name_v, input string fld_name_v, input uvm_reg_data_t val_v);

function void set_field_compare(input string rg_name, input string fld_name, input uvm_check_e check);

function uvm_check_e get_field_compare(input string rg_name, input string fld_name);

function uvm_reg_data_t get_field_data(input string register_name, input string field, uvm_reg_data_t reg_data);

function uvm_reg_data_t get_field(input string register_name, input string field);

task update_mirror(input string rg_name, input string fld_name);

function void mirror_field(input string rg_name_v, input string fld_name_v, uvm_check_e check_v);

function void status_regs_handle(input string rg_name, input string fld_name, regs_cmd_t cmds, uvm_check_e check, bit [15:0]
    val);

task predict_mirror_field_after_busy(); //processa gli oggetti nella mailbox

task handle_predict_mirror_after_busy(); //mette le predizioni nella mailbox
```

Queste sono tutte funzioni che servono a gestire i registri(visti nel capitolo due) fornendo funzioni di base per l'update e/o il mirroring dei valori.

7.3 Funzioni da implementare

Ci sono due funzioni che devono essere implementate: *get_reg_by_name* e *default_map*.

```
extern virtual function uvm_reg get_reg_by_name(string reg_name);
extern virtual function uvm_reg_map default_map();
```

Quando viene creato uno scoreboard è necessario dichiarare una variabile che contiene i registri che di solito è passata dall'ambiente fino allo Scoreboard.
Queste due funzioni vanno implementate utilizzando le rispettive funzioni con lo stes-

so nome (se il blocco dei registri estende *uvm_reg_block* contiene queste funzioni), altrimenti se vengono richiamate all'interno di un delle funzioni sopra verrà lanciato un warning senza però restituire il valore necessario.

7.4 Gestione della FSM

Nello scoreboard viene anche gestita la macchina a stati finiti e può essere inizializzata al suo interno come abbiamo visto nel capitolo precedente.
Molto spesso viene anche utilizzata al suo interno magari inducendo dei cambi di stato o semplicemente utilizzato lo stato per fare delle operazioni.

8 Ambiente parallelo

8.1 Introduzione

In questo capitolo facciamo un riassunto di tutto quello che si è parlato, specificatamente creiamo un esempio di ambiente di verifica partendo dai file contenuti nella libreria e vedendo come queste classi possono essere connesse fra di loro.

Precisiamo che questo è un ambiente che ”coesiste” in parallelo: significa che c’è un

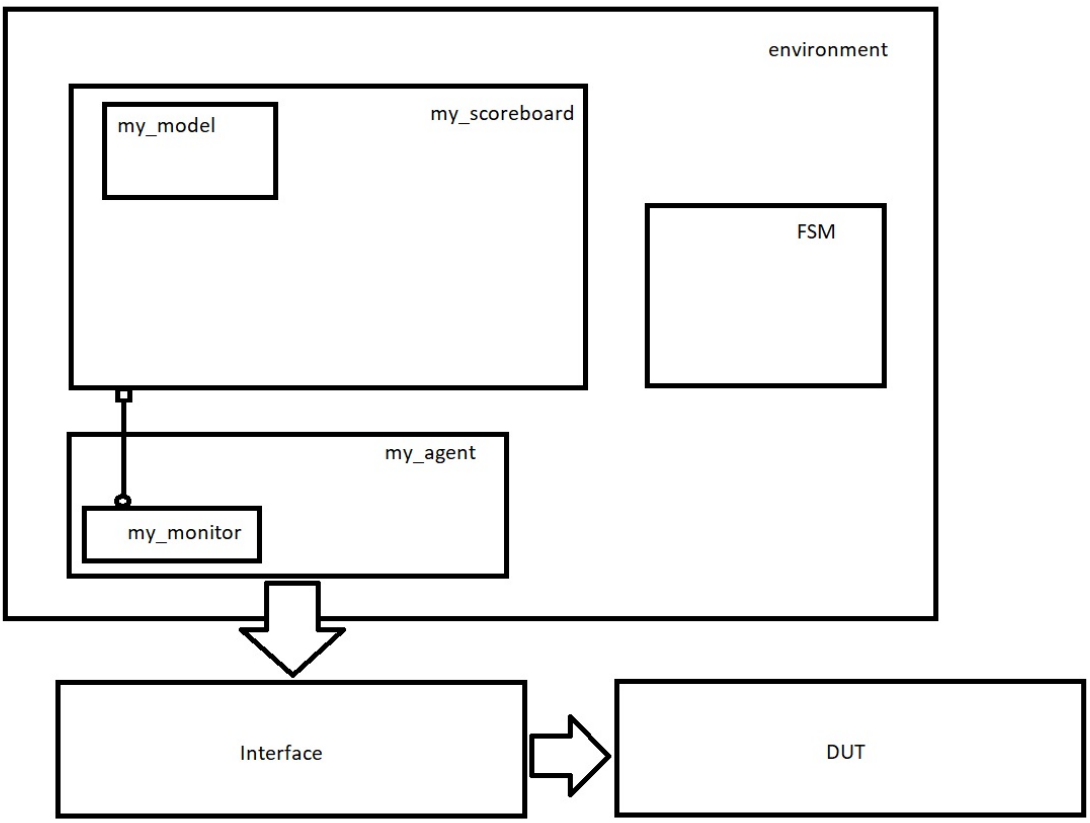

Figura 8.1: Ambiente parallelo

altro ambiente, quello già presente per testare il circuito, che è sempre presente e ne viene aggiunto un altro per testare più a fondo la libreria.
Una cosa che forse non è chiara e che è meglio specificare è che tutte le classi che sono state create non solo vengono utilizzate nell'ambiente che verrà creato in questo capitolo ma sono state introdotte anche in quello di partenza.
Per la macchina a stati finiti è stato introdotto un esempio nel capitolo 4 e quindi non verrà riportato nuovamente il codice: il funzionamento è lo stesso, cambiano solo gli stati che saranno presenti.

8.2 Il mio modello

Per esempio: supponiamo che ho un modello molto semplice e che devo monitorare solo sbalzi di tensione, può essere facilmente complicato aggiungendo il monitoring per corrente e/o temperatura facendo delle modifiche.

```
class my_model extends ifx_tb_model_base#(abist_kind_t);

  `uvm_component_utils(my_model)

  function new(string name="my_model", uvm_component parent);
    super.new(name,parent);
  endfunction

  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    fork
      check_ov_rise();
      check_uv_rise();
      check_stg_rise();
    join
  endtask: run_phase

  virtual function void ov_potent_flt_react();
    sev_flt_react(TRK1_OV,1, VALID);
  endfunction

  virtual function void uv_potent_flt_react();
    sev_flt_react(TRK1_UV,1, VALID);
  endfunction

  virtual function void stg_potent_flt_react();
    sev_flt_react(TRK1_STG,1, VALID);
  endfunction
endclass
```

Nel codice riportato sopra faccio un monitoring degli eventi di rise/fall che verranno triggherati dallo scoreboard richiamando la funzione *set_signal_ov()* che contiene la macro corrispondente.

8.3 Lo scoreboard

Adesso analizziamo lo scoreboard, il modello è contenuto al suo interno però questo non è necessario. In alternativa posso creare un componente che contiene tutti i modelli

e passare questo allo scoreboard utilizzando la libreria UVM.

```
`uvm_analysis_imp_decl(_monitor_pkt)

class my_scoreboard extends ifx_tb_scoreboard;
  `uvm_component_utils(my_scoreboard)

  uvm_analysis_imp_monitor_pkt#(ifx_base_seq_item#(monitor_en_source_t,bit[23:0]),my_scoreboard) scoreboard_port;
  my_model mod;

  function new(string name="my_scoreboard", uvm_component parent);
    super.new(name,parent);
    scoreboard_port=new("imp_port",this);
    mod=my_model::type_id::create("my_model",this);
  endfunction

  virtual function void write_monitor_pkt(ifx_base_seq_item#(monitor_en_source_t,bit [23:0]) item);
    //lancia un trigger al monitor
    case(monitor_en_source_t'(item.source))
      TRK1_MON:begin
          `uvm_info(get_type_name(),$sformatf("Packet received from TRK1: %0s",item.convert2string()),UVM_LOW);
          `uvm_info(get_type_name(),$sformatf("Type of fault: %0p",voltage_t'(item.data)),UVM_LOW);
          case(voltage_t'(item.data))
            OV:mod.set_signal_uv(1, item.kind);
            UV:mod.set_signal_ov(1, item.kind);
            STG:mod.set_signal_stg(1, item.kind);
          endcase
        end
    endcase
  endfunction
endclass
```

Come possiamo vedere la funzione *write_monitor_pkt()* viene richiamata quando viene mandato un dato di un tipo specifico attraverso la porta, questo proviene dal monitor che verrà collegato solitamente a livello di environment.
Al suo interno se arriva un pacchetto che ha come provenienza *TRK1_MON* viene segnalato al modello che c'è stato un errore e questo fa in modo che la funzione di check, che è sempre in ascolto, lanci la routine per gestire l'errore.

8.4 Il monitor

Adesso vediamo la creazione del monitor, la cosa più importante è che bisogna aggiungere le variabili che permettono di monitorare i tempi e forniscono le tempistiche sul tipo di segnale(*GLITCH*,*POSSIBLE* e *SURE*).

```
class monitor_trks extends ifx_base_monitor#(monitor_en_source_t,bit[23:0]);

  `uvm_component_utils(monitor_trks)
  extern function new(string name, uvm_component parent);
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void connect_phase(uvm_phase phase);
  extern virtual task run_phase(uvm_phase phase);

  extern protected task monitor();

endclass: monitor_trks

task monitor_trks::run_phase(uvm_phase phase);
  fork
    monitor();
  join
endtask: run_phase

task automatic monitor_trks::monitor();

```

```
21   fork
22         monitors_en = 1;
23       ana_monitor_cover();
24       monitor_voltage(TRK1_MON, vif.uv_trk1_mon_h_i, vif.ov_trk1_mon_h_i);
25   join
26 endtask: monitor
```

Bisogna inoltre abilitare la coverage chiamando la funzione *ana_monitor_coverage()* e mettere un monitor di tensione sul *TRK1*.

8.5 L'AGENT

Una volta fatto questo è necessario creare un driver e un agent. Per quanto riguarda il primo viene utilizzato quello che è stato modificato e inserito nell'ambiente di base. L'agent viene fornito qui di seguito, semplicemente crea il monitor e lo connette tramite la porta che poi a livello di ambiente è connessa allo scoreboard.

```
1  class agent_trks extends uvm_agent;
2
3    `uvm_component_utils(agent_trks)
4
5    // analysis port connected to monitor
6    uvm_analysis_port #(ifx_base_seq_item#(monitor_en_source_t,bit [23:0])) ana_monitor_port;
7    ifx_base_monitor_config t_config;         // handle to configuration object
8    ana_monitor_pkg::ana_monitor_driver t_driver;         // signal driver
9    monitor_trks t_monitor;       // signal monitor
10   ana_monitor_pkg::ana_monitor_sequencer t_sequencer;  // transaction sequencer
11   virtual interface ana_monitor_if ana_mon_if;
12
13   extern function new(string name , uvm_component parent);
14   extern virtual function void build_phase(uvm_phase phase);
15   extern virtual function void connect_phase(uvm_phase phase);
16 endclass: agent_trks
17
18 function agent_trks::new(string name, uvm_component parent);
19   super.new(name, parent);
20 endfunction : new
21
22 function void agent_trks::build_phase(uvm_phase phase);
23   super.build_phase(phase);
24
25   // costruisce la porta
26   ana_monitor_port = new("ana_monitor_port", this);
27
28   `uvm_info(get_name(),$sformatf("Building ANA UVC %p Agent",t_config.is_active),UVM_LOW)
29
30   t_monitor = monitor_trks::type_id::create("t_monitor", this);
31   t_monitor.t_config = t_config;
32   //connette le interfacce
33   t_monitor.vif = ana_mon_if;
34   t_monitor.checks_enable = t_config.checks_enable;
35   t_monitor.coverage_enable = t_config.coverage_enable;
36
37   if(t_config.is_active == UVM_ACTIVE) begin
38     t_driver = ana_monitor_pkg::ana_monitor_driver::type_id::create("t_driver", this);
39     t_driver.drv_buff_size = t_config.drv_buff_size;
40     t_sequencer = ana_monitor_pkg::ana_monitor_sequencer::type_id::create("t_sequencer", this);
41   end
42
43 endfunction: build_phase
44
45 function void agent_trks::connect_phase(uvm_phase phase);
46   super.connect_phase(phase);
47   t_monitor.monitor_port.connect(ana_monitor_port);
48
49   // connect sequencer and driver (if existing) sequence item ports
50   if(t_config.is_active == UVM_ACTIVE) begin
51     t_driver.vif = ana_mon_if;
52     t_driver.seq_item_port.connect(t_sequencer.seq_item_export);
53     t_driver.rsp_port.connect(t_sequencer.rsp_export);
```

```
54   end
55 endfunction: connect_phase
```

Viene anche connesso il driver al sequencer perchè le sequenze che verranno date devono essere interfacciate con il dispositivo. Si ricorda che tutto parte dal sequencer che invia i dati al driver che fa da tramite fra l'ambiente ed il dispositivo sotto test.

8.6 L'AMBIENTE

Nell'ambiente è necessario creare lo scoreboard e l'agent e connettere questi due tramite una porta.
In questo caso vengono inoltre caricate due variabili che contengono la configurazione del monitor e quella dell'ambiente, che contiene l'interfaccia, che ad un livello superiore dovranno essere create e caricate utilizzato la funzione set del configuration database.

```
1  class my_env extends uvm_environment;
2
3    `uvm_component_utils(my_env)
4
5    my_scoreboard my_scb;
6    agent_trks ana_trks_agent;
7    ana_config ana_cfg;
8
9    function void new(string name="my_env",uvm_component parent=null);
10     super.new(name,parent);
11     my_scb = my_scoreboard::type_id::create("my_scb", this);
12     ana_trks_agent = agent_trks::type_id::create("agent_trks", this);
13   endfunction
14
15   function void build_phase(uvm_phase phase);
16     super.build_phase(phase);
17     //passa il file di configurazione al monitor
18     if (!uvm_config_db #(ifx_base_monitor_config)::get(this, "*", "ana_trks_cfg", ana_trks_cfg))
19     `uvm_fatal(get_name(), "Get Configuration Object ana_trks_cfg not successful in ana_env.");
20
21     //passa il file di configurazione dell'ambiente
22       if (!uvm_config_db #(ana_config)::get(this, "*", "ana_cfg", ana_cfg))
23     `uvm_fatal(get_name(), "Get Configuration Object ana_cfg not successful in ana_env.");
24
25     ana_trks_agent.t_config = ana_trks_cfg;
26     ana_trks_agent.ana_mon_if=ana_cfg.ana_mon_if;
27   endfunction
28
29   function void ana_env::connect_phase(uvm_phase phase);
30     super.connect_phase(phase);
31     ana_trks_agent.ana_monitor_port.connect(my_scb.scoreboard_port);
32   endfunction
33 endclass
```

Nell'immagine successiva ci sono delle forme d'onda che rappresentano lo stato del dispositivo e il disturbo introdotto.
Come si può vedere quando il monitor si accorge che il segnale è diventato un *SURE*, viene mandato un oggetto allo scoreboard tramite la porta che a sua volta notifica il modello e fa scattare un cambio di stato da *INIT* a *FAILSAFE*.

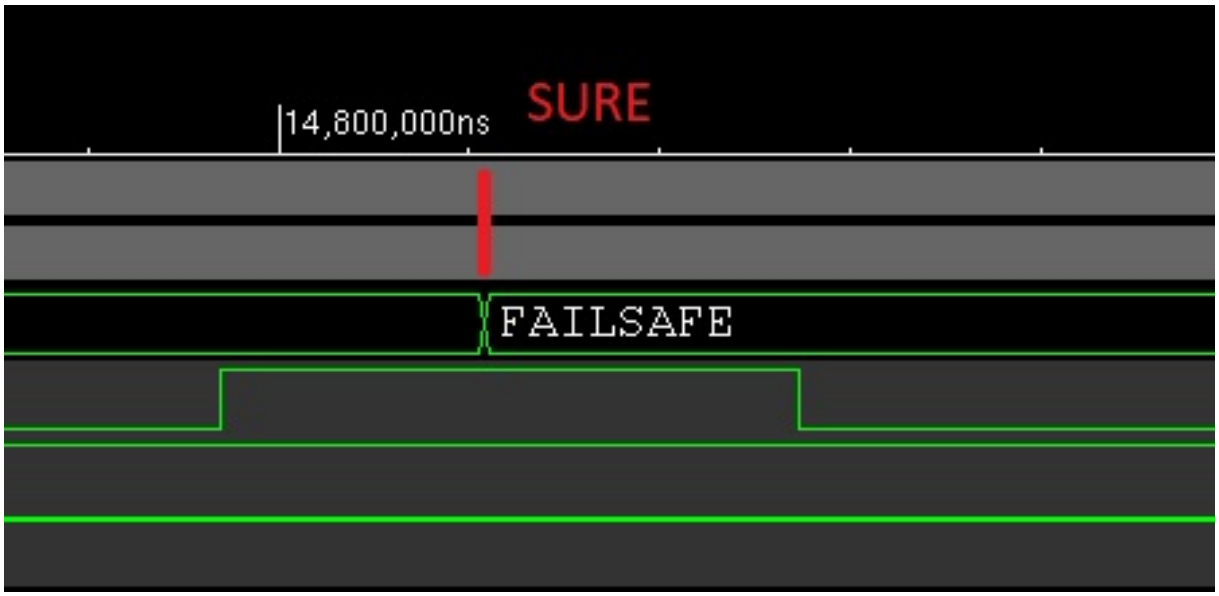

Figura 8.2: Introduzione di un fault

9

Conclusioni e sviluppi futuri

Nello sviluppo del progetto l'obbiettivo era quello di creare un insieme di classi che facilitassero la creazione di un ambiente di test con lo scopo di riutilizzare il codice e renderlo il più generico possibile.
Inoltre ci si era imposti di utilizzare questo insieme di classi in un progetto reale per testarne il funzionamento.
Date queste premesse nella tesi è stato descritto il funzionamento nello specifico dei vari blocchi che si possono utilizzare spiegando i passi necessari all'utilizzo di tale libreria.
Il progetto è stato inoltre inserito in un ambiente di test per la verifica digitale di un PMIC sviluppato da Infineon Italia.
Si può inoltre notare che questo progetto potrebbe essere la base di un progetto più grande nel quale si potrebbero includere molte più classi con scopi specifici sviluppati da altri gruppi all'interno dell'azienda e di rendere questa libreria usufruibile a tutti i dipendenti.

10 Bibliografia

Le due fonti utilizzate per studiare il linguaggio System Verilog e la libreria UVM sono riportate qui di seguito.

[1] System Verilog for Verification - A Guide to Learning the Testbench Features
Chris Spear-Greg Tumbush - 2012

[2] https://www.chipverify.com/uvm/uvm-tutorial