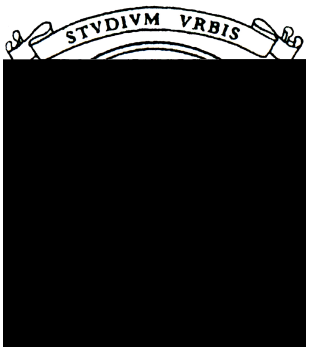


Università degli studi di Roma "La Sapienza"

Facoltà di Ingegneria

A.A. 2006/2007

Corso di laurea in Ingegneria Informatica

Tesina per il corso di

Metodi Formali nell'Ingegneria del Software

Verifica hardware: progettazione ed implementazione di uno strumento automatico per la verifica formale delle proprietà di semplici reti digitali

A cura di:

Claudio Corona

Giorgio Camerani

Abstract

L'obiettivo del presente lavoro è quello di progettare e realizzare uno strumento automatico che permetta di effettuare la verifica di semplici reti digitali[1]: con il termine *verifica* si vuole intendere, in questa sede, quell'attività il cui obiettivo consiste nel dimostrare formalmente che la rete logica in esame soddisfi alcune particolari proprietà. Tali proprietà possono essere di tipo *generale*, come ad esempio la certezza di terminazione della computazione o l'assenza di loopback combinatori, oppure di tipo *specifico*, vale a dire non generalizzabili, in quanto strettamente legate agli obiettivi e alle caratteristiche del particolare sistema digitale oggetto della verifica. Alcuni esempi di proprietà *specifiche* potrebbero essere i seguenti:

- Progettando un sistema digitale per il monitoraggio di un apparato elettrocardiografico, vorremmo essere certi che, nei casi in cui il valore della frequenza cardiaca del paziente fuoriesca da un opportuno intervallo di tolleranza, il sistema emetta effettivamente un segnale sonoro di allarme;

- Nel caso di un sistema digitale per il controllo di un insieme di semafori presenti ad un incrocio stradale, potremmo voler dimostrare matematicamente l'inesistenza di una qualsivoglia evoluzione del sistema che sia tale da compromettere le normali condizioni di sicurezza del traffico (ad esempio, il sistema non deve mai poter rendere verdi nello stesso momento le luci di due semafori).

- Volendo progettare un sistema digitale per l'arbitraggio di un bus dati, una proprietà desiderabile è senz'altro la seguente: non deve mai essere permesso a due dispositivi differenti di accedere contemporaneamente in scrittura al bus (analogia con il problema della mutua esclusione);

- Nel caso di un sistema per l'elaborazione numerica di un segnale (o di un insieme di segnali), potremmo voler dimostrare che il sistema in esame si comporti effettivamente come il filtro desiderato, vale a dire che effettui sul segnale le operazioni e le trasformazioni per le quali è stato progettato.

E' chiaro che l'elenco potrebbe continuare: le possibilità potenziali di utilizzo sono pressochè infinite. Si noti come i sistemi digitali oggetto della verifica possano spaziare dalla *computazione* vera e propria (intesa come semplice implementazione di algoritmi di calcolo, come ad esempio addizionatori, moltiplicatori, filtri per segnali) al *controllo* (inteso come governo e orchestrazione di più dispositivi, come ad esempio implementazione di protocolli

[1] Utilizzeremo i termini *rete digitale* e *rete logica* come sinonimi.

di handshaking, monitoraggio di segnali biomedici, gestione di macchine industriali operanti in modo sincronizzato).

Si osservi come, dopo quanto appena detto, risulti evidente una interessante possibilità da prendere in considerazione nel presente lavoro: una volta che si sarà progettato un tool di verifica automatica *sufficientemente generale*, si potrà dare libero spazio alla creatività, senza alcun limite di sorta. In altre parole, se il tool che verrà sviluppato sarà *generale* al punto da poter permettere la descrizione di una *qualsivoglia* rete logica, è chiaro che esso sarà di valido aiuto tanto nella verifica di proprietà generali, quanto nella verifica di proprietà specifiche: la distinzione tra proprietà generali e specifiche non avrà più alcuna rilevanza, nel senso che, dal punto di vista del tool, le due tipologie saranno indistinguibili (mentre rimarranno separate e distinguibili dal punto di vista soggettivo del progettista che effettua la verifica).

In definitiva, dunque, l'obiettivo centrale è il seguente: permettere all'utente di descrivere una qualunque rete logica[2], assemblandola a partire dalle sue componenti elementari (in ultima istanza, infatti, qualunque rete digitale, per complessa che sia, è costituita a partire da un ristretto numero di componenti fondamentali, quali flip-flop e porte logiche combinatorie), per poi effettuare su di essa la verifica di alcune proprietà di interesse, specificate di volta in volta dal progettista.

La traduzione automatica avverrà nel seguente modo: dal momento che ogni rete logica R *è* una macchina a stati in evoluzione al trascorrere del tempo (scandito dal clock), la descrizione di R verrà convertita nella descrizione di una struttura temporale a stati, da fornire in input all'applicazione NuSMV. Fatto ciò, dovrà essere possibile interrogare NuSMV circa alcune particolari proprietà di interesse della rete R: tali proprietà verranno espresse in logica temporale lineare (LTL). Un'opportunità interessante potrebbe essere quella di fornire all'utente una sintassi intermedia, che gli consenta di esprimere le proprietà della rete digitale utilizzando sì gli operatori di LTL, ma con alcune semplificazioni che gli risparmino di dover conoscere la pletora di variabili generate dalla procedura di traduzione automatica: in altre parole, si otterrebbe un buon risultato se si riuscisse a randere completamente trasparente la presenza, dietro le quinte, di NuSMV. In un simile scenario, le proprietà da verificare verrebbero quindi espresse direttamente all'interno del tool di conversione, il quale si occuperebbe della traduzione dell'espressione da verificare in pura sintassi LTL, invocando poi silenziosamente l'applicazione NuSMV. E' chiaro che, a tal punto, per chiudere il cerchio, sarebbe necessario

[2]Nel caso in cui il carico di lavoro risultasse eccessivo, si potrebbe semplicemente prevedere l'importazione di file generati da opportuni strumenti software, espressamente dedicati al disegno di circuiti logici.

che il nostro tool analizzasse anche l'output fornito da NuSMV, effettuando un mapping inverso dalla struttura temporale alla rete R: ad esempio, nel caso in cui NuSMV mostrasse un cammino nel quale la proprietà in esame non è soddisfatta dalla rete R, sarebbe interessante mostrare tale cammino in forma grafica, visualizzandolo in termini di una computazione della rete (mostrando il contenuto dei vari registri passo dopo passo, secondo l'evoluzione temporale indicata da NuSMV).

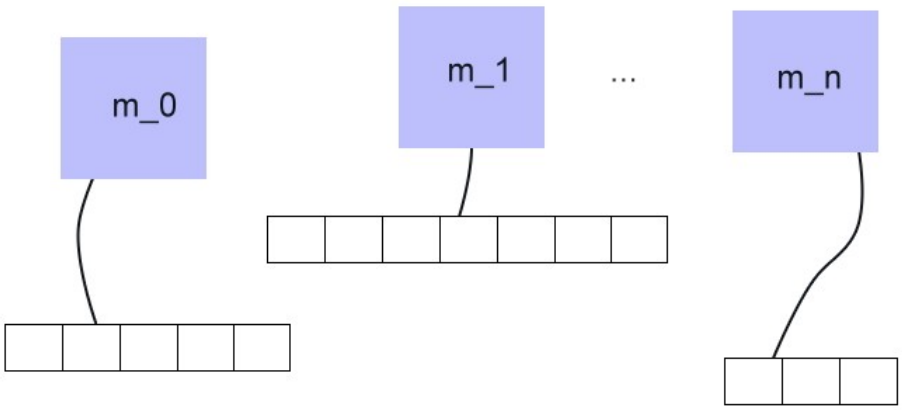
Da quanto detto, emerge l'esigenza di progettare ed implementare il tool con il *massimo grado di generalità* possibile.

Il parsing

L'attività di parsing dei file di tipo Verilog (estensione .vlg) viene effettuata da una opportuna classe, VerilogParser.java. Lo standard Verilog prevede la presenza di un certo numero di componenti primitivi, quali ad esempio le porte logiche elementari. In questa sede, gli unici componenti primitivi che ci interessano sono le porte AND, OR, NOT, NAND, NOR, XOR, XNOR, il flip-flop di tipo D, la tensione positiva (equivalente, logicamente, al valore 1, TRUE) e la tensione di massa (equivalente, logicamente, al valore 0, FALSE): essi saranno sufficienti per la costruzione di una *qualsivoglia* rete logica. La struttura di un generico file Verilog è la seguente:

```
module m0 ( g_{m0,0}, g_{m0,1}, ..., g_{m0,T(m0)} )
       input g_{m0,1}, g_{m0,2}, g_{m0,3}, ..., g_{m0,I(m0)};
       output g_{m0,1}, g_{m0,2}, g_{m0,3}, ..., g_{m0,O(m0)};
       wire w_{m0,1}, ..., w_{m0,W(m0)}
       type_{c0} c0 ( w_{c0,1}, w_{c0,2}, w_{c0,3}, ..., w_{c0,T(c0)} )
       ...
       type_{cK(m0)} cK(m0) ( w_{cK(m0),1}, w_{cK(m0),2}, w_{cK(m0),3}, ..., w_{cK(m0),T(cK(m0))} )
end module

module m1 ( g_{m1,0}, g_{m1,1}, ..., g_{m1,T(m1)} )
       ...
end module

...

module mn ( g_{mn,0}, g_{mn,1}, ..., g_{mn,T(mn)} )
       ...
end module
```

Come si vede, nel file sono dichiarati n moduli, ognuno dei quali ha $T(m_i)$ porte: di queste, $I(m_i)$ sono ingressi e $O(m_i)$ sono uscite (ovviamente $T(m_i) = I(m_i) + O(m_i)$). La parola chiave wire fa riferimento ai nodi del circuito: tuttavia, le informazioni contenute in righe come `wire w_{m0,1}, ..., w_{m0,W(m0)}` non sono di interesse per il parsing, nel senso che si può fare a meno di esse. Ogni modulo m_i fa uso, al suo interno, di $K(m_i)$ componenti: è chiaro un modulo può fare uso di più componenti dello stesso tipo. L'attività di parsing ha l'effetto di costruire una struttura dati che ricalca esattamente la struttura del file Verilog, appena illustrata; la seguente figura mostra tale struttura dati:



Ogni modulo è legato ad un vettore di componenti da esso utilizzati. Nel seguito utilizzeremo i termini modulo e componente come sinonimi. Si noti come nel file Verilog non sia presente alcuna indicazione su quale sia il componente principale del circuito: esso deve essere determinato dalla classe Verilogparser.java (è sufficiente individuare quale sia il componente che non viene mai

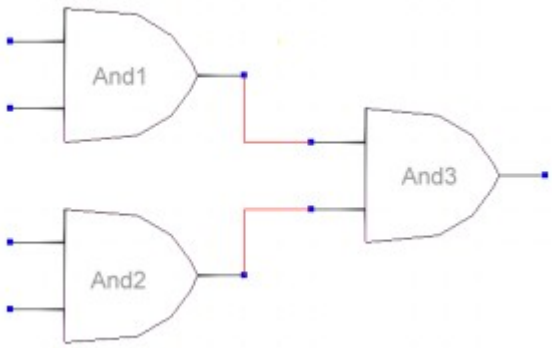
utilizzato da nessun altro componente: sarà proprio esso il componente principale). A questo punto, illustriamo brevemente il meccanismo utilizzato per la rappresentazione e la determinazione delle connessioni tra i vari componenti del circuito. La precedente illustrazione mostra ad alto livello la struttura dati utilizzata per la rappresentazione del circuito, ma non coglie alcuni fondamentali dettagli. Il fatto che il modulo (sinonimo di componente, ma continuiamo ad utilizzare la parola "modulo" per evidenziare quei componenti la cui descrizione è esplicita nel file Verilog) m_0 utilizzi, ad esempio, il componente and1 di tipo porta logica AND non è di per sé sufficiente per informarci su come quel componente sia internamente connesso nel modulo m_0. Per questo motivo è stata predisposta, all'interno di ciascun modulo m_i, una struttura tabellare che ne rappresenti le connessioni interne tra i componenti costituenti. Tale tabella è costituita da 5 colonne e da tante righe quanti sono i fili di interconnessione: in altre parole, una simile tabella è un insieme di *link*, dove un *link* è definito come una quintupla $<C_I, i, C_O, o, w>$. Prima di spiegare cosa rappresentano gli elementi di tale quintupla, illustriamo un breve quanto intuitivo concetto, utile per capire il razionale che ha condotto alla scelta di simili quintuple. Il concetto è il seguente: dal momento che ogni componente presente in un circuito è interpretabile come una scatola nera avente k ingressi ed m uscite, possiamo dire che un qualsivoglia collegamento (filo) tra due componenti mette in comunicazione l'output di un componente con l'input di un altro componente. Si faccia attenzione ai due casi particolari: si può mettere in comunicazione l'input di un componente con l'input di un altro componente, oppure si può mettere in comunicazione l'output di un componente con l'output di un altro componente. Nel primo caso si sta semplicemente dicendo che gli input dei due componenti saranno alimentati dallo stesso output, mentre nel secondo caso si sta affermando un qualcosa di patologico per un circuito: connettere due output insieme significa infatti introdurre un cortocircuito (sarà interessante accorgersi di come, all'atto della traduzione del circuito in un file ".smv", NuSMV sarà in grado di rilevare automaticamente simili situazioni di cortocircuito, riconoscendole come errori semantici di assegnazione multipla). Fatta questa breve premessa, la spiegazione del *link* (quintupla) di cui sopra risulta facile ed immediata:

C_I: componente che riveste il ruolo di utilizzatore del segnale (input)
i: indice dell'ingresso di C_I cui si riferisce il collegamento in questione
C_O: componente che riveste il ruolo di fornitore del segnale (output)
o: indice dell'uscita di C_O cui si riferisce il collegamento in questione
w: variabile usata nel file Verilog per denotare il filo che realizza il collegamento in questione

L'ultimo elemento della quintupla, la variabile w, non fornisce informazioni finali né riveste alcun ruolo nella successiva traduzione nel formato ".smv". Tuttavia, essa è assolutamente fondamentale durante il processo di *costruzione* della tabella dei collegamenti (serve per trovare i matching, cioè per determinare la presenza di un collegamento: si noti infatti come nel file Verilog i collegamenti non siano espressi esplicitamente con una struttura apposita, ma l'informazione ad essi relativa è "dispersa" tra i vari componenti). Vediamo un esempio:

C_I	i	C_O	o	w
And3	0	And1	0	w1
And3	1	And2	0	w2

Con la tabella soprastante, stiamo dicendo che il componente And3 riceve nel suo primo ingresso (0) il primo output del componente And1, e nel suo secondo ingresso (1) il primo output del componente And2. L'immagine relativa ai collegamenti della tabella di cui sopra è la seguente (assumiamo che i componenti And1, And2 e And3 in gioco siano tutti di tipo porta logica AND):



Riassumendo, la struttura dati rappresentata nella precedente immagine, arricchita dalla tabella appena descritta, permette di ricostruire esattamente *qualunque* circuito di partenza.

La conversione in formato .smv

Descriviamo ora il cuore dell'intero elaborato: la conversione della struttura dati appena illustrata in un file .smv che ricalchi esattamente la topologia del circuito, e che permetta di verificarvi alcune interessanti proprietà di natura dinamica. Di tale conversione si occupa la classe VerilogToSMVConverter.java. Si noti come la struttura a moduli insita nel file Verilog si presti molto semplicemente ed elegantemente ad una immediata trasposizione in altrettanti moduli di NuSMV. Ciò è esattamente quello che viene fatto: data la struttura dati rappresentante il generico circuito (immagine di pag.1), vengono prodotti tanti moduli NuSMV quanti sono i moduli m_i della struttura dati (ricordiamo che tali moduli sono quelli esplicitamente definiti come tali nel file Verilog). Per ognuno di tali moduli NuSMV vengono predisposte alcune variabili:

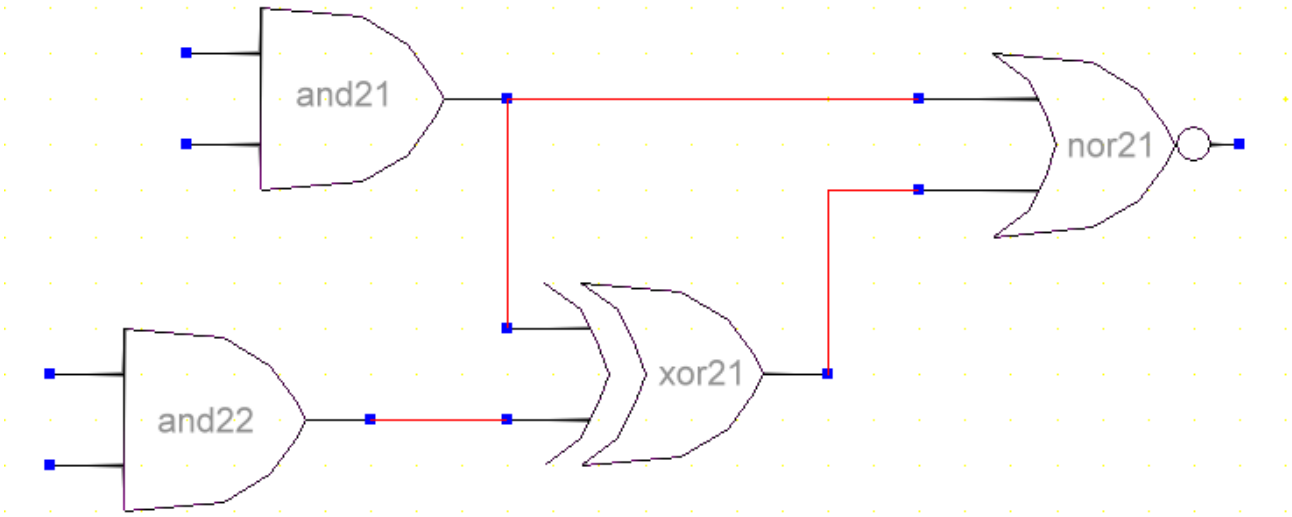
1. Una variabile per ogni ingresso (input),
2. Una variabile per ogni uscita (output),
3. Una variabile per ogni componente utilizzato.

Per quanto riguarda il punto 3, è necessaria una precisazione. Se un modulo m_i fa uso, al suo interno, di un certo componente c, tale componente c può essere primitivo (AND, OR, NOT, Flip-Flop D, ...) oppure no. Nel caso c sia primitivo, è chiaro che nel file Verilog (e quindi nella struttura dati assemblata a partire da esso) non ci sarà esplicita descrizione del suo funzionamento interno: la sua funzione caratteristica è assunta essere già nota implicitamente. Per questa ragione è chiaro che il processo di conversione dovrà riconoscere quali sono i componenti primitivi e quali quelli complessi: se si incontra il nome di un componente primitivo, la sua descrizione (nota a priori) dovrà essere aggiunta al file .smv che stiamo costruendo; se invece si incontra il nome di un componente complesso (cioè che non figura tra i nomi di quelli primitivi), si sa che (se il file Verilog è corretto) prima o poi troveremo un modulo con lo stesso nome che descrive il funzionamento interno di tale componente complesso. In altre parole: nel file Verilog i componenti primitivi non vengono descritti, mentre nel file .smv vogliamo che lo siano.
Il riconoscimento del modulo principale, brevemente descritto in precedenza, è molto importante nel processo di conversione, da momento che permette di individuare quale debba essere il modulo `main` all'interno del file .smv: tale modulo `main` farà uso di tutti gli altri moduli. Prima di illustrare quanto detto finora con un esempio, è utile mostrare il modo in cui sono stati mappati i collegamenti (fili) dalla struttura dati in memoria (tabella dei collegamenti di cui sopra) al file .smv. Cerchiamo di capire cosa esprime, dal punto di vista delle variabili, la presenza di un filo nel circuito. Osservando l'immagine 2, vediamo come l'uscita della porta AND And1 sia collegata al primo ingresso della porta AND And3. Immaginiamo il file .smv relativo al semplice circuito nell'immagine 2: esso presentarà 1 modulo `and` e 1 modulo `main`: il modulo `main` farà uso di 3 variabili di tipo `and`, una per ogni porta logica AND in gioco (And1, And2, And3).

Il generico modulo `and` avrà al suo interno 3 variabili: 2 per gli ingressi (in_1, in_2) e 1 per l'uscita (out). Ora, con i fili rappresentati nell'immagine 2 stiamo dicendo che:

- La variabile in_1 della porta And3 è costretta (vincolata) ad essere uguale alla variabile out della porta And1,
- La variabile in_2 della porta And3 è costretta (vincolata) ad essere uguale alla variabile out della porta And2.

Ecco quindi cosa esprimono, dal punto di vista delle variabili di ingresso e di uscita, i fili di un circuito: essi sono dei vincoli, che impediscono ad un certo ingresso di mutare a piacimento, costringendolo ad essere sempre uguale, in ogni istante di tempo t, al valore dell'uscita ad esso collegata. Riassumiamo quanto detto finora con un esempio chiarificatore. Supponiamo di avere il seguente circuito:



Il file Verilog corrispondente al circuito soprastante è il seguente:

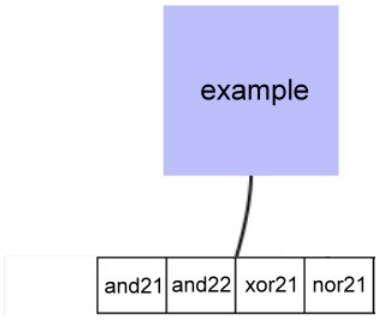
```
module example();
 wire w1,w2,w3,w4,w5,w6,w7,w8;
 wire w9,w10;
 and and21(w8,w6,w7);
 and and22(w5,w3,w4);
 xor xor21(w9,w8,w5);
 nor nor21(w10,w8,w9);
endmodule
```

Vi è un solo modulo (quello principale), privo di input e di output esterni. Si noti come i collegamenti tra componenti siano ricostruibili a partire dagli argomenti dei vari componenti. Ad esempio la riga:

```
and and21(w8,w6,w7);
```

afferma la presenza di un componente di tipo `and`, chiamato `and21`. I suoi argomenti sono `w8,w6,w7` : il primo è relativo all'output, il secondo è relativo al primo input e il terzo è relativo al secondo input. In sostanza gli argomenti w_i rappresentano alcuni nodi nel circuito: quando troviamo lo stesso w_i nella lista di argomenti di due componenti differenti, sappiamo che esiste un filo tra quei due componenti. Con riferimento a `w8` vediamo che esso figura anche come secondo argomento del componente `xor21` e come secondo argomento del componente `nor21`: ecco quindi che grazie a `w8`

rileviamo la presenza di 2 fili: il primo va dall'output di `and21` al primo input di `xor21`, e il secondo va sempre dall'output di `and21` al primo input di `nor21`. La struttura dati descritta in precedenza, relativamente al presente esempio, è la seguente:



C_I	*i*	C_O	*o*	*w*
xor21	0	and21	0	w8
xor21	1	and22	0	w5
nor21	0	and21	0	w8
nor21	1	xor21	0	w9

Infine, il file .smv generato per questo esempio è il seguente:

```
MODULE and
      VAR
            in_1  :     boolean;
            in_2 :      boolean;

      DEFINE
            out   :=    in_1 & in_2;

MODULE xor
      VAR
            in_1  :     boolean;
            in_2 :      boolean;

      DEFINE
            out   :=    ( in_1 & !in_2 ) | ( !in_1 & in_2 );

MODULE nor
      VAR
            in_1  :     boolean;
            in_2 :      boolean;

      DEFINE
            out   :=    !( in_1 | in_2 );

MODULE main
      VAR
            and21 :     and;
            and22 :     and;
            nor21 :     nor;
            xor21 :     xor;

      ASSIGN
            xor21.in_1  :=    and21.out;
            xor21.in_2  :=    and22.out;
            nor21.in_1  :=    and21.out;
            nor21.in_2  :=    xor21.out;
```

Qualche commento sul codice .smv è necessario per chiarire la gestione dei componenti primitivi e per chiudere il discorso sui collegamenti tra componenti. Si noti come i moduli AND, XOR e NOR presenti nel circuito, essendo primitivi, non vengano descritti internamente nel file Verilog, mentre invece vengano descritti nel file .smv. Ciò avviene perchè la loro descrizione è sì implicita nello standard Verilog, ma noi dobbiamo pur istruire NuSMV su come tali componenti si comportano. Si noti l'uso del costrutto sintattico DEFINE, che permette di definire una nuova variabile in funzione

delle altre già presenti. Si noti infine il modo in cui viene "implementata" la considerazione fatta in precedenza sui collegamenti: abbiamo detto che un collegamento (filo) impone un vincolo alla variabile che nel collegamento riveste il ruolo di input, costringendola ad essere sempre uguale alla variabile che gioca il ruolo di output. Un simile vincolo si traduce, in NuSMV, facendo uso del costrutto ASSIGN: esso permette di assegnare ad una variabile un certo valore (eventualmente di un altra variabile, come avviene in questo caso). Si osservi come non sia presente, nelle varie assegnazioni, né la parola chiave `init`, né la parola chiave `next`. Infatti, effettuare una assegnazione del tipo:

```
var1  :=    var2;
```

vuol dire imporre il vincolo che `var1` sia *sempre* uguale a `var2`, non solo nell'istante iniziale. In altre parole, qualunque cammino nella struttura temporale sarà vincolato a contenere solo stati in cui l'assegnazione di cui sopra risulta vera. Notiamo infine come una simile assegnazione sia anche esprimibile mediante la seguente coppia di assegnazioni:

```
init(var1)  :=    init(var2);
next(var1)  :=    next(var2);
```

I componenti sequenziali

Finora, nell'illustrare il processo di parsing e di conversione di un generico circuito, abbiamo fatto riferimento, negli esempi, a soli componenti di natura combinatoria. E' chiaro come, affinchè un circuito esibisca un comportamento dinamico, evolutivo, sia necessario inserire al suo interno dei componenti sequenziali, cioè con memoria, che permettano di memorizzare lo stato corrente del circuito, e quindi di far evolvere la computazione anche sulla base della storia passata (riepilogata nello stato corrente). E' evidente, inoltre, che la verifica di proprietà interessanti con LTL richiede, di per sé, la presenza del concetto di cambiamento di stato del circuito (in realtà, alcune proprietà interessanti possono essere dimostrate anche su circuiti interamente combinatori, tuttavia sia l'utilità sia l'interesse didattico aumentano notevolmente quando si verificano reti sequenziali).
L'uso di componenti di natura sequenziale è stato progettato nel seguente modo: si è predisposto un unico componente primitivo di natura sequenziale: il flip-flop D (si tratta della medesima scelta presente nello standard Verilog). *Ogni altro componente sequenziale è stato assemblato a partire unicamente da flip-flop D e da eventuali componenti combinatori.* E' stato interessante, poi, testare i componenti sequenziali più complessi così assemblati, per verificare che essi fossero conformi al loro comportamento teorico. Tutto questo si è tradotto in una affascinante eleganza del listato .smv risultante: le uniche transizioni presenti sono quelle che descrivono il comportamento di un generico flip-flop D.
Prima di vedere il codice .smv relativo ad un generico modulo flip-flop D, è opportuno fare alcune precisazioni sulle scelte progettuali che hanno guidato la successiva implementazione:

- LTL non prevede minimamente la presenza del concetto di lunghezza dell'intervallo temporale tra due istanti: non possiamo dire se tra un istante e il successivo sono passati 3 millisecondi o 1 nanosecondo. Semplicemente ci viene offerto il concetto del fluire del tempo, modellato come sequenza discreta di istanti, equidistanti temporalmente (questo limite viene superato, ad esempio, dalla logica TCTL). Per questa ragione, non si è tenuto conto della frequenza di clock: si assume che ogni istante *coincida con l'arrivo di un fronte di clock* (pertanto i componenti sequenziali di cui parliamo si intendono *edge-triggered*). Inoltre, per quanto riguarda i componenti combinatori, viene da sé che essi effettueranno le loro computazioni *istantaneamente* (tempo nullo): se in un certo stato un componente combinatorio c presenta determinati valori per i suoi ingressi, allora i corrispondenti valori per le sue uscite sono già pronti in quel medesimo stato (essendo frutto di vincoli di tipo DEFINE), senza aspettare l'istante successivo.

- Si è deciso di *non* modellare la presenza, nei componenti sequenziali, di ingressi di reset asincrono. Ciò non rappresenta un limite problematico, dal momento che è possibile costruire reti digitali complesse, verificando su di esse proprietà interessanti, anche se non si hanno a disposizione ingressi di reset asincrono.

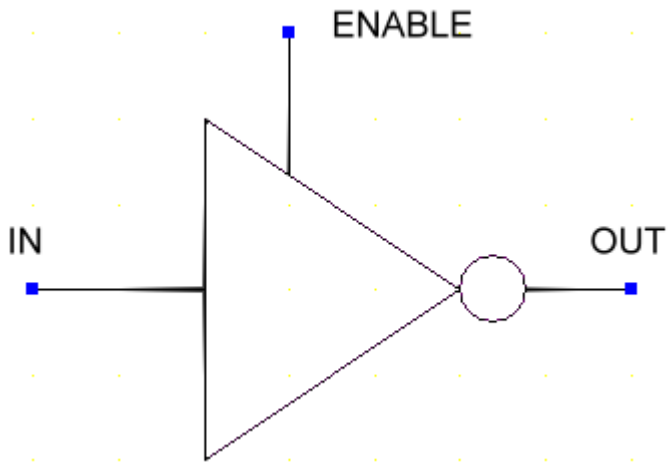
Veniamo ora alla descrizione dell'unico componente primitivo di tipo sequenziale: il flip-flop D. All'interno dei file Verilog, esso viene chiamato dreg (D Register), e così viene anche chiamato nei file .smv prodotti dalla conversione. Esso è un componente primitivo all'interno dei file Verilog, nel senso che non ne viene descritto il comportamento: noi però dobbiamo descrivere tale comportamento, a beneficio di NuSMV. Il modulo NuSMV relativo ad un generico flip-flop D è il seguente:

```
MODULE dreg
      VAR
            d     :     boolean;
            q     :     boolean;

      DEFINE
            nq    :=    !q;

      TRANS
            case
                  d = 1 :     next(q) = 1;
                  d = 0 :     next(q) = 0;
            esac
```

Si noti la semplicità e l'eleganza delle transizioni. E' davvero sorprendente accorgersi di come, con queste due elementari transizioni, si possano assemblare reti logiche di arbitraria complessità.
Non c'è altro da dire in merito ai componenti sequenziali, dal momento che essi vengono tutti sintetizzati a partire dal flip-flop D. Negli esempi successivi, mostreremo la sintesi e la verifica di componenti sequenziali quali il flip-flop T, il flip-flop JK, lo shift-register e il contatore.

L'inverter 3-state

Descriviamo ora il modo in cui viene tradotto nella sintassi NuSMV l'inverter 3-state. Esso merita un discorso a parte a causa della sua natura sui-generis. Vedremo come la soluzione adottata per modellare il suo funzionamento sia di affascinante semplicità, ed in linea con l'eleganza globale finora mostrata dalla conversione. Ricordiamo che l'inverter 3-state è un componente che presenta 2 ingressi e 1 uscita:

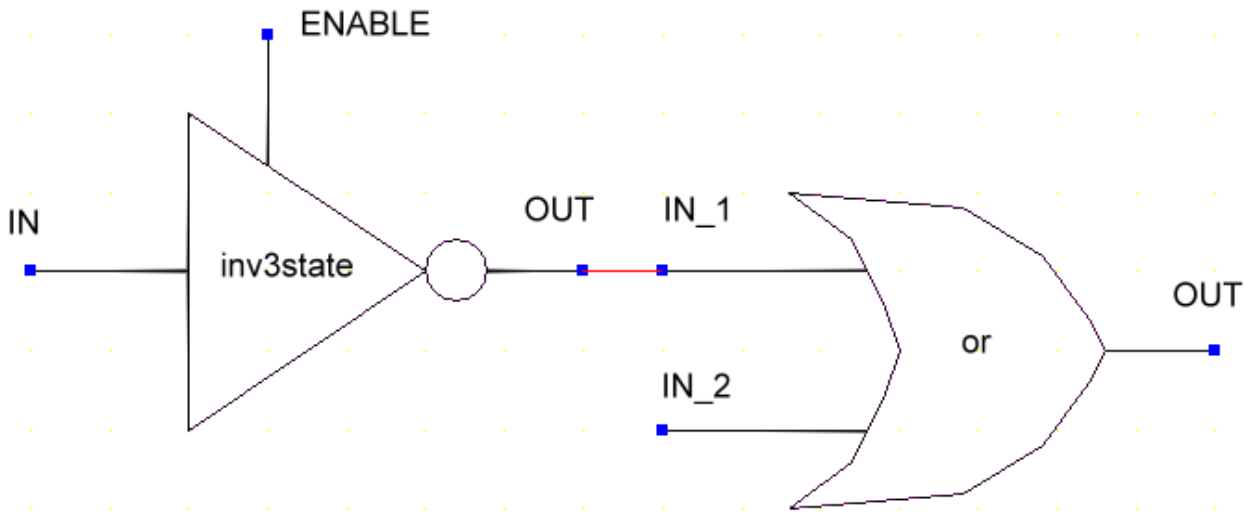


Il suo funzionamento è il seguente: se l'ingresso ENABLE è pari a TRUE, allora OUT è pari al negato di IN (inversione dell'ingresso); se invece l'ingresso ENABLE è pari a FALSE, allora OUT è

nello stato di *alta impedenza* (Z). Dire che OUT è in alta impedenza significa dire che esso è libero di assumere un qualunque valore tra TRUE e FALSE: non è vincolato dall'ingresso, e risente di forme d'onda nelle vicinanze (si comporta come un'antenna). Ora, è chiaro che un simile comportamento non è lineare, dal momento che presenta sia una semantica di tipo if-then-else sia la presenza di un valore, Z, non appartenente all'insieme dei valori di verità canonici. Il fatto che tale comportamento non sia lineare comporta l'impossibilità di esprimere l'uscita OUT come *funzione* degli ingressi, usando il solo simbolo di uguaglianza. *Si deve far uso dell'implicazione*.
Il modulo NuSMV relativo al generico inverter 3-state è il seguente:

```
MODULE inv3state
      VAR
            input       :     boolean;
            enable      :     boolean;
            output      :     { TRUE, FALSE, z };

      INVAR
            ( enable ->
                  ( ( input = TRUE -> output = FALSE ) &
                  ( input = FALSE -> output = TRUE ) )
            ) &
            ( !enable -> output = z )
```

Si noti il dominio della variabile output, comprendente, oltre ai canonici valori di verità TRUE e FALSE, anche il valore z (alta impedenza). Il punto importante è nel blocco INVAR: il costrutto INVAR serve per esprimere una formula in logica proposizionale che deve valere sempre, durante tutta l'evoluzione temporale del modulo (trattasi appunto di una *invariante*). Stiamo dunque definendo il comportamento dell'inverter 3-state non più definendo una nuova variabile (come avveniva nel caso delle porte combinatorie elementari, usando il costrutto DEFINE), bensì con una formula, *sempre vera*, che ne vincoli l'evoluzione. La formula è semplice: se l'ingresso ENABLE è a TRUE, allora l'uscita OUT è *è costretta* ad essere pari al negato dell'ingresso IN; se invece l'ingresso ENABLE è a FALSE, allora l'uscita *è costretta* ad essere pari a z. Fin qui abbiamo modellato il comportamento interno dell'inverter 3-state: restano ora da modellare correttamente i collegamenti che gli ingressi di altri generici componenti possono avere con l'uscita OUT dell'inverter 3-state. *Non è più possibile utilizzare, come nei generici casi sopra esposti, il costrutto ASSIGN per modellare i collegamenti, per le stesse ragioni appena illustrate.* Simili collegamenti vanno quindi modellati facendo uso del costrutto INVAR; chiariamo il tutto con un esempio:



L'ingresso IN_1 della porta OR è collegato all'uscita OUT dell'inverter 3-state. Non possiamo utilizzare, come in precedenza, la seguente assegnazione:

```
or.in_1          :=    inv3state.out;
```

In tal caso, infatti, NuSMV rileva giustamente un errore semantico: stiamo assegnando ad una variabile (`or.in_1`) il valore di un'altra variabile (`inv3state.out`) *avente un dominio più ampio.* Infatti la variabile `or.in_1` può assumere i soli valori TRUE e FALSE, mentre la variabile `inv3state.out` può assumere i valori TRUE, FALSE e z. La soluzione è usare il costrutto INVAR, adoperando una formula contenente una opportuna implicazione, come illustrato qui sotto:

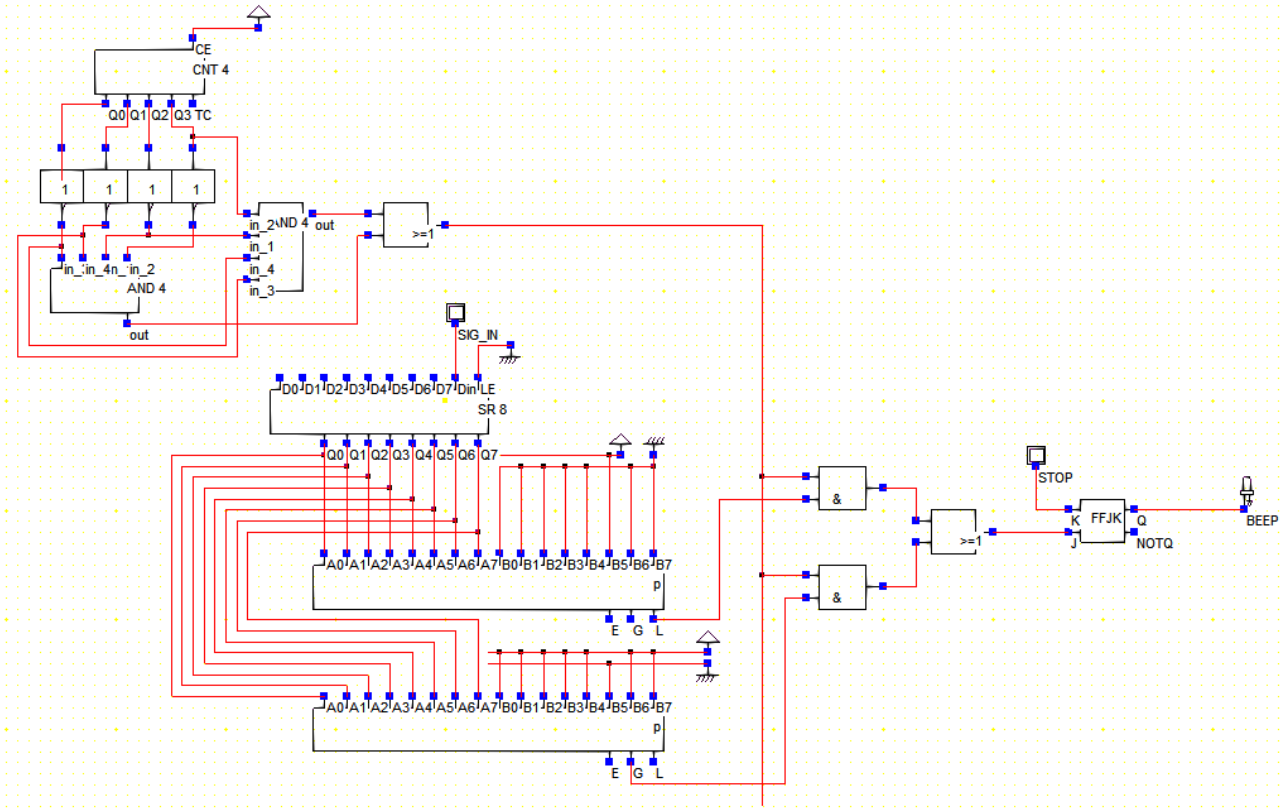
```
INVAR
      ( inv3state.out = TRUE -> or.in_1 = TRUE ) &
      ( inv3state.out = FALSE -> or.in_1 = FALSE )
```

Stiamo vincolando l'ingresso IN_1 della porta OR ad essere pari all'uscita OUT dell'inverter 3-state solo se tale uscita *è diversa da z*. *Qualora l'uscita OUT dell'inverter 3-state sia pari a z, le premesse delle due implicazioni risultano false, e le due implicazioni risultano vere: l'invariante è quindi già soddisfatta in partenza, e l'ingresso IN_1 della porta OR è libero di portarsi ad uno qualunque dei due valori TRUE o FALSE*. Ciò ricalca *esattamente* il comportamento teorico descritto in precedenza: l'ingresso IN_1 della porta OR si comporta come una antenna, essendo libero di oscillare in modo aleatorio e imprevedibile tra TRUE e FALSE.

La possibilità di prevedere uno SCO

Illustriamo molto brevemente come sia possibile, con grande semplicità, prevedere la presenza di uno SCO (Sistema di COntrollo) che funga da direttore d'orchestra per il generico circuito progettato. Il generico circuito nella sua interezza sarà certamente dotato di m ingressi e k uscite: esso è quindi assimilabile, ad alto livello, ad una scatola nera provvista di input (per influenzarne il comportamento) e di output (per leggerne l'evoluzione). Ricordiamo che il generico circuito nella sua interezza è rappresentato, nel file .smv, dal modulo `main`. Per inserire uno SCO, sarà sufficiente prevedere un nuovo modulo `sco`: all'interno del modulo `main`, poi, verrà definita una variabile di tipo `sco` e, con opportune assegnazioni, verranno collegati gli output del `main` agli input dello `sco`, e gli output dello `sco` agli input del `main`: in questo modo lo `sco` leggerà lo stato del circuito dalle sue uscite e ne influenzerà l'evoluzione agendo sui suoi ingressi. Si ottiene così un mutuo accoppiamento tra i due sistemi (SCA e SCO): essi evolvono influenzandosi a vicenda. E' chiaro, poi, che il modulo `sco`, al suo interno, sarà dotato di opportune transizioni che ne ricalcheranno la struttura algoritmica di controllo (ciò è ottenibile abbastanza facilmente a partire dal diagramma di flusso dell'algoritmo di controllo, secondo la medesima metodologia di traduzione dei programmi imperativi).

Esempio – Monitoraggio del segnale elettrocardiografico



Vediamo qui un esempio di rete logica complessa: trattasi di un circuito per il monitoraggio del segnale elettrocardiografico. Il segnale arriva in modo seriale attraverso l'ingresso SIG_IN e viene immagazzinato all'interno di uno shift-register a 8 bit, che permette di trasformare il segnale seriale in segnale parallelo (SIPO, Serial In Parallel Out), a blocchi di 8 bit (si assume che il generico valore della frequenza cardiaca del paziente richieda 8 bit). Il contatore serve per verificare che siano passati 8 istanti, e che quindi lo shift-register contenga un dato significativo (altrimenti si tratta di dati transitori). Due comparatori si occupano di controllare che il valore non ecceda l'intervallo considerato sicuro (il valore della frequenza cardiaca deve trovarsi tra 32 e 224). Se ciò accade, e se il valore corrente è significativo, viene abilitato l'ingresso J di un flip-flop JK, che abilita l'uscita Q (connessa al segnale BEEP) fin quando un dispositivo esterno (uno SCO) non provvede a resettare il flip-flop JK con il segnale STOP (connesso all'ingresso K del flip-flop JK). La verifica formale di questo circuito è consistita nel dimostrare che se il valore della frequenza cardiaca eccede uno dei limiti di soglia allora viene emesso un BEEP, e che se viene emesso un BEEP allora vuol dire che ciò è accaduto perchè è stato ecceduto uno dei limiti di soglia. Si tratta di una verifica di doppia implicazione (se e solo se): in particolare, il secondo verso è dimostrabile facendo uso di un opportuno operatore per il passato (l'operatore O, once, duale di F, supportato da NuSMV).