UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
Facoltà di Scienze, Matematiche, Fisiche e Naturali
Corso di Laurea triennale di Informatica

TESI DI LAUREA

IMPLEMENTAZIONE DI UN ALGORITMO DI VERIFICA FORMALE PER PROGRAMMI GERARCHICI NEL TOOL YASM

Relatore: Prof. Aniello Murano

Candidato: Vadim Malvone
Matricola: 566/002835

Anno Accademico 2009/2010

Indice

1 Introduzione....3
2 Il processo di Model Checking....7
2.1 Modellazione del sistema....7
2.2 Specifica....11
2.2.1 La logica LTL....11
2.2.2 La logica CTL....14
2.3 Verifica....19
2.3.1 Soddisfacibilità e model checking....19
2.3.2 Model checking esplicito....20
2.4 Model checking basato su automi....21
2.4.1 Automa di Büchi non-deterministico su parole infinite....21
2.4.2 Automa di Büchi alternato su parole infinite....22
2.4.3 Trasformazione di una formula in LTL ad automa....23
2.4.4 Model checking basato su automi in LTL....24
2.4.5 Automa non-deterministico su alberi binari infiniti....25
2.4.6 Automa alternato su alberi infiniti....26
2.4.7 Model checking basato su automi in CTL....27
3 Il problema dell'esplosione degli stati....29
3.1 Model checking simbolico....29
3.1.1 Alberi binari di decisione....30
3.1.2 Binary Decision Diagram....31
3.1.3 Ordered Binary Decision Diagrams....32
3.1.4 Rappresentare Strutture di Kripke....35
3.2 Model checking astratto....37
3.3 Model checking gerarchico....43
3.3.1 Struttura di Kripke gerarchica....44

3.4 I model checker....47
4 Yasm: il tool analizzato e le modifiche sviluppate....53
4.1 Prima esecuzione sul tool Yasm....53
4.2 Analisi del tool Yasm....58
4.2.1 Descrizione dettagliata del PProgram....61
4.2.2 Ulteriori considerazioni sull'analisi....65
4.2.3 Descrizione dettagliata del CFA....66
4.2.4 Descrizione di una XKripkeStructure....68
4.3 Descrizione dell'obbiettivo e modifiche effettuate....70
4.3.1 pProgram gerarchico....70
4.3.2 CFA gerarchico....73
4.3.3 Ultime modifiche effettuate....77
5 Conclusioni....80

1 Introduzione

Tratteremo in questa tesi gli aspetti relativi alla verifica formale dei sistemi e, in particolare, quella dei sistemi software.
La verifica formale è uno strumento molto importante per analizzare la correttezza di sistemi hardware. Il problema della correttezza dei sistemi è un problema particolarmente sentito, specialmente nel contesto dei sistemi critici.
Quando parliamo di sistemi critici intendiamo tutti quei sistemi il cui fallimento non è assolutamente accettabile. I principali sistemi critici si dividono in :

- safety, in cui il fallimento può costare vite umane;
- mission, in cui il fallimento può costare in termini di obbiettivi;
- business, in cui il fallimento può costare in perdite di denaro.

Negli ultimi decenni, abbiamo avuto diversi esempi di sistemi critici importanti che hanno avuto comportamenti inattesi con conseguenze disastrose. Negli anni ottanta, ad esempio, diverse persone sono state uccise e altre gravemente ferite da un cattivo malfunzionamento del sistema Therac-25, una macchina per irradiazione usata in campo medico. Nello specifico, il cattivo funzionamento è stato prodotto da errori nel software e in un errato interfacciamento tra hardware e software.
Nel 1995 il nuovissimo aeroporto di Denver progettato per essere all'avanguardia con il suo complesso sistema computerizzato per lo smistamento dei bagagli è stato costretto ad aprire 16 mesi dopo la data prevista per difetti del sistema. Il cattivo funzionamento del software ha portato ad uno sforamento delle spese per 3,2 milioni di dollari.
Nel 1996 il vettore Arianne 5 fu lanciato nello spazio, punta di diamante del programma spaziale europeo. Dopo 36.7 secondi di volo il sistema ha deviato la rotta, esplodendo subito dopo. Il disastro è stato causato da un programma del

sistema di navigazione che ha tentato di mettere un numero di 64 bit in una spazio da 16 bit. Il costo del danno è stato di circa un milione di euro, oltre al danno di immagine!

Nel 1997 una nave da guerra americana comandata da un avanzatissimo sistema di gestione computerizzata basato su Windows NT è rimasta paralizzata in mare per due ore. Successivamente, si è scoperto che la paralisi è stata dovuta da un buffer overflow. Fortunatamente per loro era solo una simulazione...

Descrivendo questi esempi ci rendiamo conto che verificare la correttezza di un sistema software prima del suo utilizzo è un aspetto fondamentale.

I principali metodi di verifica del software sono:

- testing;
- simulazione;
- verifica deduttiva;
- verifica formale o model checking.

Il testing sul software è un operazione effettuata durante tutta la fase di sviluppo in cui il prodotto viene controllato per trovare eventuali malfunzionamenti e in tal caso per risolverli prima del rilascio.

La simulazione è simile al testing solo che il processo di verifica viene eseguito su un modello astratto del sistema. Lo svantaggio principale è che non vi sono garanzie che tutte le esecuzioni possibili vengano simulate.

Entrambi i metodi descritti sopra possono rilevare gli errori ma non possono stabilirne l'assenza.

La verifica deduttiva utilizza assiomi e regole di dimostrazione per verificare la correttezza del sistema. Questo tipo di metodo è decisamente dispendioso per quanto riguarda il tempo ed è quindi utilizzato principalmente per i sistemi altamente sensibili.

La verifica formale è una tecnica automatica che, dato un sistema ed una proprietà, valuta formalmente se la proprietà è verificata nel sistema (figura 1-1).

Una tecnica di verifica formale di grande successo è il model checking. In questo caso, il sistema viene convertito in un modello formale e la verifica è demandata ad un tool chiamato *model checker* che verifica la correttezza del sistema "via modello". Quando parliamo di model checking di solito il modello è rappresentato da una struttura di Kripke, ovvero, un grafo orientato in cui ogni nodo rappresenta una possibile configurazione del sistema e ogni arco rappresenta una possibile transizione da una configurazione ad un'altra.

Nel model checking, la proprietà, anche detta specifica, viene solitamente data in una formula di qualche logica temporale. Le logiche temporali sono particolarmente utili nella verifica formale dei sistemi perché possono descrivere l'ordine degli eventi senza introdurre il tempo esplicitamente. In questa tesi noi utilizzeremo LTL e CTL, che saranno ampiamente descritte nel prossimo capitolo.

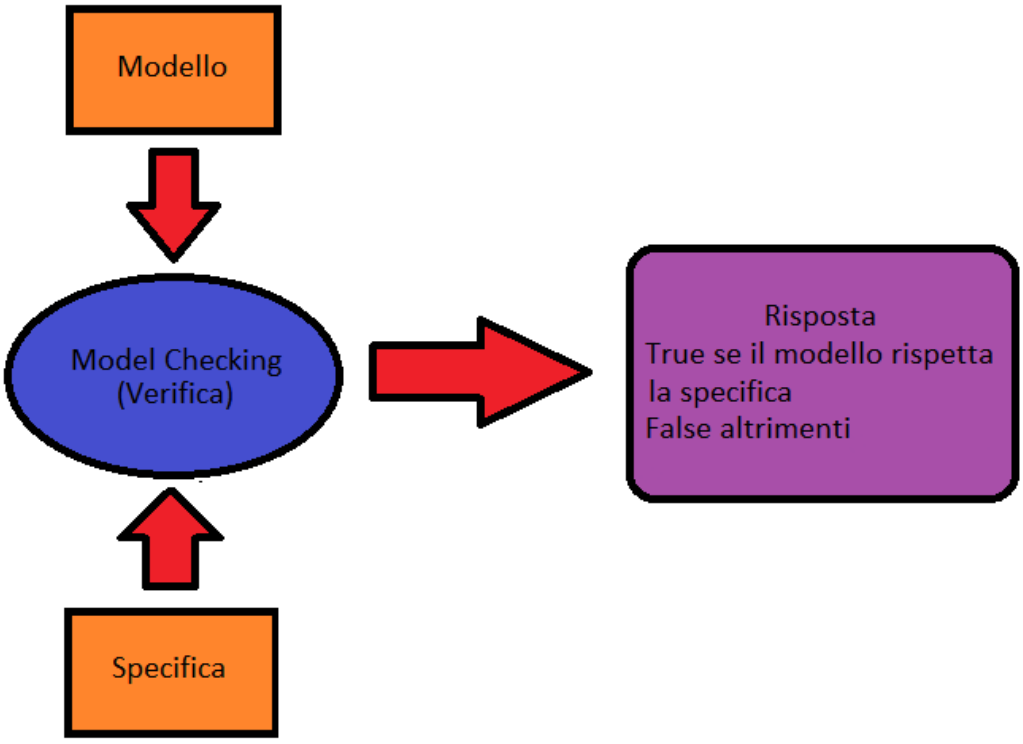

Figura 1-1

Nel capitolo 3 tratteremo il problema dell'esplosione degli stati. Questo problema nasce dal fatto che gli stati della struttura di Kripke rappresentano le possibili evoluzioni delle variabili del sistema sotto esame. E' evidente che nella modellazione, questi stati sono in numero elevatissimo anche per piccoli software. Il problema dell'esplosione degli stati è solitamente il collo di bottiglia nell'uso del model checking per la verifica dei sistemi.
Nei capitolo 3 analizziamo tre metodi per ridurre questo problema:

- verifica simbolica;
- tecniche di astrazione;
- modello gerarchico.

La verifica simbolica è un metodo in cui il modello non è rappresentato esplicitamente ma viene descritto da strutture dati speciali chiamate BDD (Binary Decision Diagram).
L'astrazione invece sfrutta la conoscenza sul modello e sulla proprietà in modo tale da modellare solo le caratteristiche rilevanti del modello.
L'ultimo metodo che analizzeremo sarà il model checking gerarchico. Questa tecnica crea un modello gerarchico i cui nodi possono essere stati o super-stati. I super-stati rappresentano a loro volta un altro modello.
Nel capitolo 4 descriveremo Yasm, un model-checker simbolico basato su CEGAR (Counter-Example Guided Abstraction Refinement) e daremo un esempio di esecuzione.
Nell'ultima parte della tesi tratteremo le modifiche effettuate sul tool Yasm per la verifica di un software gerarchico, modellato dapprima con un sistema gerarchico e poi “appiattito“ in un modello di Kripke classico utilizzando un algoritmo noto in letteratura.

2 Il processo di Model Checking

Il model checking [4] è una tecnica di verifica che permette di verificare la corretta di un sistema rispetto ad una specifica data, verificando formalmente che un modello del sistema sia corretto rispetto ad un modello della specifica. Dunque, il processo di model checking si fonda su tre fasi fondamentali e distinte: modellazione del sistema, specifica delle proprietà da analizzare, verifica formale di correttezza sul modello.

2.1 Modellazione del sistema

Nel model checking, un sistema per essere valutato ha bisogno di essere trasformato in un modello formale.
Un modello viene rappresentato come un sistema di transizioni, cioè un grafo orientato con nodi e archi. Ogni nodo rappresenta uno stato del sistema e gli archi specificano un' alterazione del sistema. Ad ogni nodo è associato un'insieme di proposizioni atomiche, che sono vere in quel nodo. Una proposizione atomica è un enunciato non scomponibile di cui si può stabilire il suo valore di verità, in altre parole se è vero o è falso [16].
Il modello formale che utilizzeremo sarà la struttura di Kripke [7].
Sia AP un insieme di proposizioni atomiche, una struttura di Kripke K è una quadrupla $K = < S, S_0, R, L >$, dove:

1. S è un insieme finito di stati;
2. $S_0 \subseteq S$ è un insieme di stati iniziali;
3. $R \subseteq S \rightarrow S$ è una relazione di transizione totale che associa ad ogni stato $s \in S$ l' insieme dei suoi successori;
4. $L : S \rightarrow 2^{AP}$ è una funzione che etichetta ogni stato con l'insieme delle proposizioni atomiche vere nello stato.

Esempio.

Presa la seguente struttura di Kripke:

- AP = { a, b };
- $S = \{w_0, w_1, w_2\}$;
- $S_0 = w_0$;
- $(w_0, w_1) \in R$, $(w_1, w_2) \in R$, $(w_2, w_0) \in R$;
- $L(w_0) = \{a\}$, $L(w_1) = \{a, b\}$, $L(w_2) = \{b\}$;

Il grafo K è il seguente :

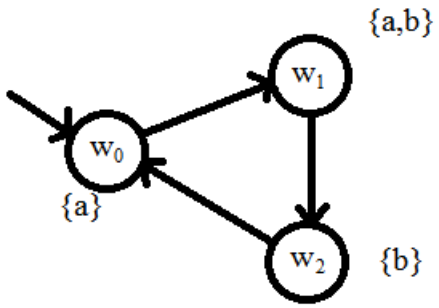

Figura 2.1-1

Preso un percorso π nel grafo K, definiamo:

$$\mathrm{Path}(K) = \{\pi \in S^S : \forall i \in N (\pi_i, \pi_{i+1}) \in R\}$$

cioè Path è l'insieme di tutti i percorsi di K.

A questo punto possiamo definire il linguaggio di una struttura di Kripke K come:

$$L(K) = \{L(\pi) : \pi \in \mathrm{Path}(K)\}$$

con $L(\pi) = L(\pi_0) * L(\pi_1) * \ldots$

Adesso possiamo analizzare il procedimento per creare da un sistema una Kripke [7].

Preso $V=\{v_1, v_2, \ldots v_n\}$ un insieme di variabili di un sistema e un dominio D di valori per le variabili di V, definiamo valutazione per V una funzione che associa ad ogni variabile $v_i \in V$ un valore di D. Quindi uno stato non è altro che una valutazione $s: V \rightarrow D$ per l'insieme delle variabili in V. A questo punto creiamo una seconda copia dell'insieme delle variabili, che chiameremo V', per rappresentare le transizioni.

Ora per derivare una Kripke K = $< S, S_0, R, L >$ non dobbiamo fare altro che associare:

- all'insieme degli stati S tutte le possibili valutazioni per V;
- all'insieme degli stati iniziali S_0 l'insieme di tutte le valutazioni s_0 per V che soddisfano una formula iniziale;
- presi due stati s e s', $R(s, s')$ vale se la formula che la rappresenta è vera quando ad ogni $v \in V$ e $v' \in V'$ sono assegnati rispettivamente i valori s(v) e $s'(v')$;
- infine la funzione di etichettatura è definita in modo che L(s) sia il sottoinsieme di tutte le proposizioni atomiche vere in s.

Prendiamo un semplice esempio di [7] per chiarire meglio. Il sistema consiste della seguente transizione: x:= (x+y) mod 2.

Definiamo gli insiemi:

- V = {x,y};
- $V' = \{x', y'\}$;
- D = {0,1};

I valori iniziali di x e y saranno entrambi 1.

Questo sistema può essere definito dalle seguenti due formule:

- formula iniziale : $x = 1 \wedge y = 1$;
- formula di transizione : $x' = (x + y) \bmod 2 \wedge y' = y$;

La strttura di Kripke K = < S, S_0, R, L > corrispondente è :

- S = { (1,1), (0,1), (1,0), (0,0) };
- S_0 = {(1,1)};
- R= { ((1,1),(0,1)), ((0,1),(1,1)), ((1,0),(1,0)), ((0,0),(0,0)) };
- L((1,1)) = { x=1,y=1}, L((0,1)) = {x=0, y=1}, L((1,0)) ={ x=1,y=0}, L((0,0)) = {x=0, y=0} ;

Ed ecco la rappresentazione grafica:

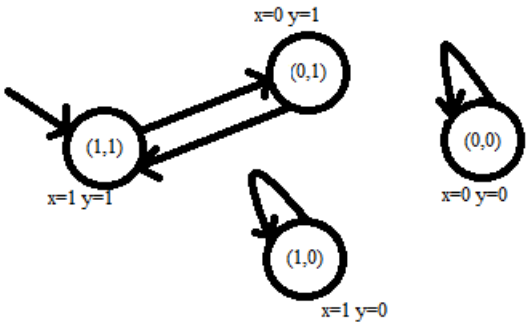

Figura 2.1-2

Notiamo facilmente grazie all'ausilio del grafo che il solo cammino che parte dallo stato iniziale è (1,1) (0,1) (1,1) (0,1) ….

2.2 Specifica

Dopo aver modellato il sistema abbiamo bisogno di rincondurre la proprietà che vogliamo esaminare ad una qualche logica.
Nel precedente paragrafo quando abbiamo parlato di formule non stavamo altro che trattando con una logica di prim'ordine, in questo caso però per esprimere una proprietà da verificare abbiamo bisogno di qualcosa in più, cioè una logica che possa descrivere l'ordine degli eventi nel tempo ma che non introduca il tempo esplicitamente [7].
Fanno al caso nostro le logiche temporali, che sono una sottoclasse delle logiche modali, nelle quali la verità di un asserzione può variare nel tempo.
Esistono vari tipi di logiche temporali, noi analizzeremo le seguenti due:

- Logica LTL (Linear Temporal Logic) ;
- Logica CTL (Computation Tree Logic) .

Queste due logiche non sono ugualmente espressive [8] , cioè nessuna delle due è inclusa strettamente nell'altra, questo ci fa capire che entrambe possono essere utili in base alla problematica in cui ci troviamo.

2.2.1 La logica LTL

La sintassi della logica LTL è la seguente [8]:

- Proposizioni atomiche;
- Connettivi Classici:
  - $\neg\psi$;
  - $\psi \wedge \varphi$;
  - $\psi \vee \varphi$;

- $\psi \supset \varphi$;

- Connettivi Temporali:
  - $F\varphi$: “φ sarà vera prima o poi”;
  - $G\varphi$: “φ sarà sempre vera”;
  - $\varphi U \psi$: “φ sarà sempre vera finchè non si verifica ψ”;
  - $X\varphi$: “ φ sarà vera nel prossimo istante”;

Oltre a questi operatori a volte ne vengono utilizzati anche altri due :

- R “release” [5] , duale a U che verifica l'equivalenza:
$$\varphi R \psi \leftrightarrow \neg((\neg \varphi) U (\neg \psi));$$
- W “weak until” [8], simile a U che verifica l'equivalenza:
$$\varphi W \psi \leftrightarrow G\varphi \vee (\varphi U \psi)$$

Dopo aver definito tutte le formule che si possono utilizzare in LTL, possiamo anche dire che lo stesso potere espressivo può essere utilizzato anche solo con gli operatori $\neg, \wedge$, X e U insieme alle preposizioni atomiche, infatti [8]:

$$F\varphi = \text{true } U \ \varphi$$
$$G\varphi = \neg F \neg \varphi$$

Adesso sviluppiamo un semplice esempio in cui possiamo rappresentare delle proprietà in logica LTL.
Vogliamo modellare le seguenti proprietà interessanti di un semaforo:

1. una volta diventato verde non può immediatamente diventare rosso;
2. prima o poi il semaforo sarà giallo;
3. una volta verde, diventa rosso dopo essere rimasto giallo per un certo periodo;

Ecco le rispettive rappresentazioni in LTL:

1. $G(\text{verde} \supset \neg X\,\text{rosso})$;
2. F giallo;
3. $G(\text{verde} \supset ((\text{verde}\ U\ \text{giallo})\ U\ \text{rosso}))$;

Sia φ una formula e K una struttura di kripke, allora la relazione $K,s \models \varphi$ (K soffisfa φ in s) è definita come [8]:

- $s \models p \leftrightarrow p \in L(s)$;
- $s \models \neg\varphi \leftrightarrow$ non vale $s \models \varphi$;
- $s \models \varphi \vee \psi \leftrightarrow s \models \varphi$ o $s \models \psi$;
- $s \models \varphi \wedge \psi \leftrightarrow s \models \varphi$ e $s \models \psi$;
- $s \models X\varphi \leftrightarrow s' \models \varphi$, dove s' è il successore di s ;
- $s \models F\varphi \leftrightarrow \exists s' \geq s : s' \models \varphi$;
- $s \models G\varphi \leftrightarrow \forall s' \geq s : s' \models \varphi$;
- $s \models \varphi U \psi \leftrightarrow \exists s' \geq s : s' \models \psi$ e $\forall s \leq s'' < s', s'' \models \varphi$;

Date sintassi e semantica possiamo adesso sviluppare un esempio.
Innanzitutto definiamo la seguente Kripke:

- AP = { p, q };
- $S = \{s_0, s_1, s_2, s_3, s_4\}$;
- $S_0 = s_0$;
- $(s_0, s_1) \in R$, $(s_1, s_2) \in R$, $(s_2, s_3) \in R$, $(s_3, s_4) \in R$, $(s_4, s_4) \in R$;
- $L(s_0) = \{\}$, $L(s_1) = \{q\}$, $L(s_2) = \{q\}$, $L(s_3) = \{q, p\}$, $L(s_4) = \{\}$;

Il grafo corrispondente è :

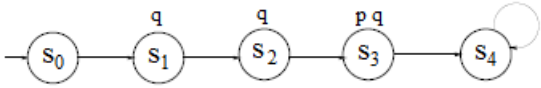

Figura 2.2.1-1

allora:

- Xp vale in s_2 ;
- Fq vale in s_0, s_1, s_2 ;
- Gp non vale in nessuno stato;
- qUp vale in s_1, s_2, s_3 :

2.2.2 La logica CTL

In LTL in ogni istante di tempo esiste un unico successore, questo è facilmente dedotto dalla sua funzione R che associa ad ogni stato un insieme costituito da uno ed un solo elemento. Da questa affermazione possiamo dire che gli operatori temporali di LTL descrivono l'ordine degli eventi lungo un singolo cammino di computazione [8] . Oltre ad LTL, come già accennato in precedenza, esiste una logica chiamata CTL in cui la nozione di tempo non è lineare bensì ha una struttura ad albero. Di seguito illustriamo le sue caratteristiche principali.
La logica CTL utilizza la seguente sintassi [8]:

- Proposizioni atomiche;
- Connettivi Classici:
  - $\neg\psi$;
  - $\psi \wedge \varphi$;
  - $\psi \vee \varphi$;

    - $\psi \supset \varphi$;
- Due tipi di connettivi:
    - Temporali: F, G, U, X;
    - Sui cammini: A, E;
- Gli operatori temporali devono essere sempre preceduti da quantificatori di cammino;
- I quantificatori di cammino devono sempre precedere gli operatori temporali;

In base alle regole stabilite dalla sintassi illustriamo gli 8 connettivi che ne derivano:

- AX φ: “per tutti i cammini di computazione la proprietà φ sarà vera nel prossimo stato”;
- EX φ: “esiste un cammino di computazione in cui la proprietà φ sarà vera nel prossimo stato”;
- AF φ: “per tutti i cammini di computazione la proprietà φ prima o poi sarà vera”;
- EF φ: “esiste un cammino di computazione in cui la proprietà φ prima o poi sarà vera”;
- AG φ: “per tutti i cammini di computazione la proprietà φ sarà sempre vera”;
- EG φ: “esiste un cammino di computazione in cui la proprietà φ sarà sempre vera”;
- A(ψ U φ): “per tutti i cammini di computazione la proprietà ψ sarà vera finchè non si verifica φ”;
- E(ψ U φ): “esiste un cammino di computazione in cui la proprietà ψ sarà vera finchè non si verifica φ”;

Ed ecco come si presentano a livello grafico [8] :

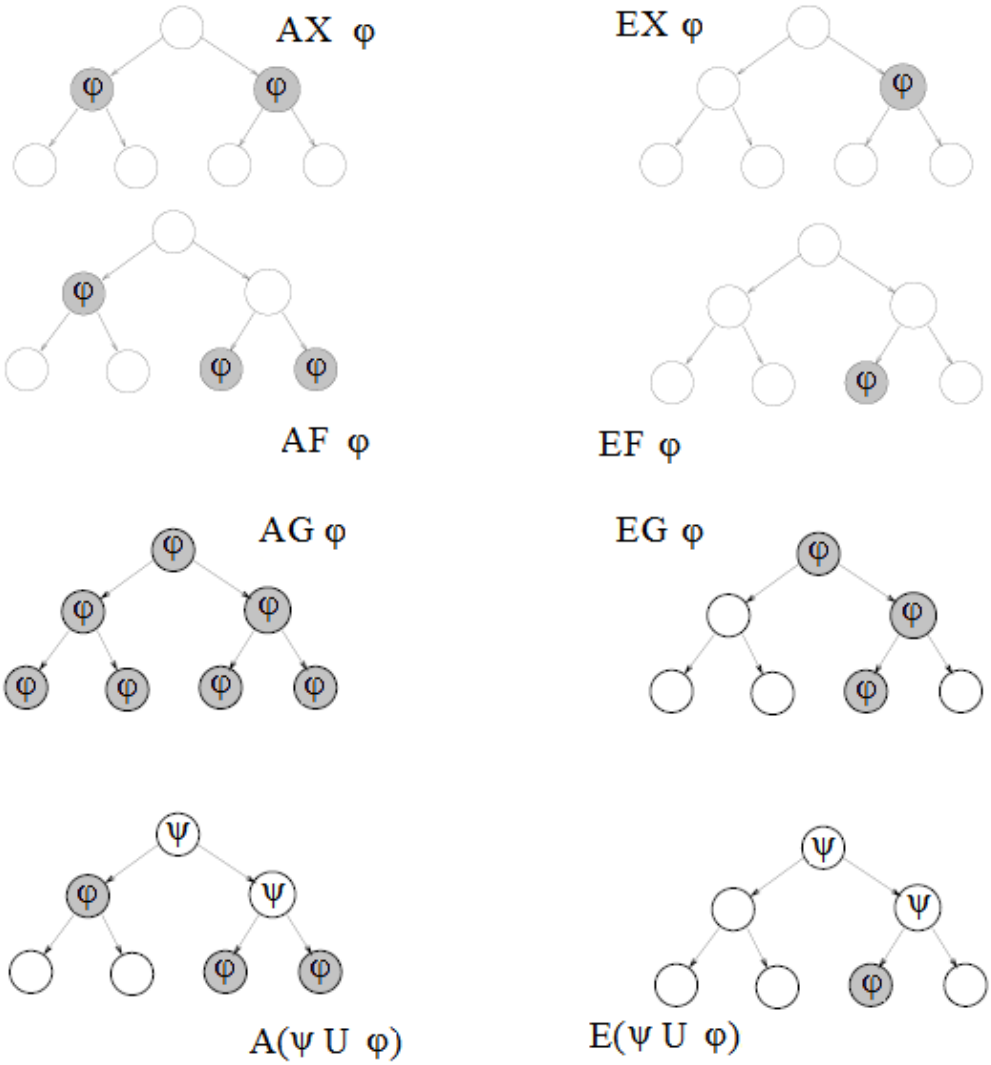

Figura 2.2.2-1

Come per LTL, anche questa logica può essere ridotta [8] al solo utilizzo degli operatori $\neg, \wedge$, EX, EG e EU insieme alle preposizioni atomiche, infatti:

$$AX\varphi = \neg EX \neg \varphi$$
$$EF\varphi = E(true\ U\ \varphi)$$
$$AG\varphi = \neg EF \neg \varphi$$
$$AF\varphi = A(true\ U\ \varphi)$$

$$A(\varphi \, U \, \psi) = \neg E[\neg\psi \, U (\neg\varphi \wedge \neg\psi)] \wedge \neg EG \neg\psi$$

Per valutare concretamente come utilizzare questi operatori, analizziamo le proprietà da rispettare in un protocollo di comunicazione con due agenti S e R come nell'esempio seguente tratto da [8] :

1. durante l'esecuzione di un sistema concorrente si può giungere ad uno stato in cui vale Start ma non Ready;
2. in ogni esecuzione ogni richiesta viene infine soddisfatta;
3. un certo processo viene abilitato infinitamente spesso in ogni esecuzione;
4. da ogni stato è possibile tornare allo stato Restart;
5. qualsiasi cosa succeda, il sistema alla fine sarà in *deadlock* permanente;

le relative formule CTL sono le seguenti:

1. $EF \, Start \wedge \neg Ready$;
2. $AG(Req \supset AF \, Ack)$;
3. AG(AF Enabled);
4. AG(EF Restart);
5. AF(AG Deadlock);

Prima di definire la semantica facciamo una piccola osservazione sui cambiamenti riguardo alla funzione R (relazione di transizione) in CTL, relativa ad una struttura di Kripke, rispetto a quella su LTL.
In CTL possono esservi più successori, questo implica che vi sono vari cammini di computazione che creano graficamente un albero(Figura 2.2.2-2).

Sia $s \in S$, una s-esecuzione σ su K è una successione infinita di elementi di S: $\sigma = s_0, s_1, s_2, \ldots$ tale che $s_0 = s$ e inoltre $R(s_i, s_{i+1})$ per ogni $i \geq 0$.

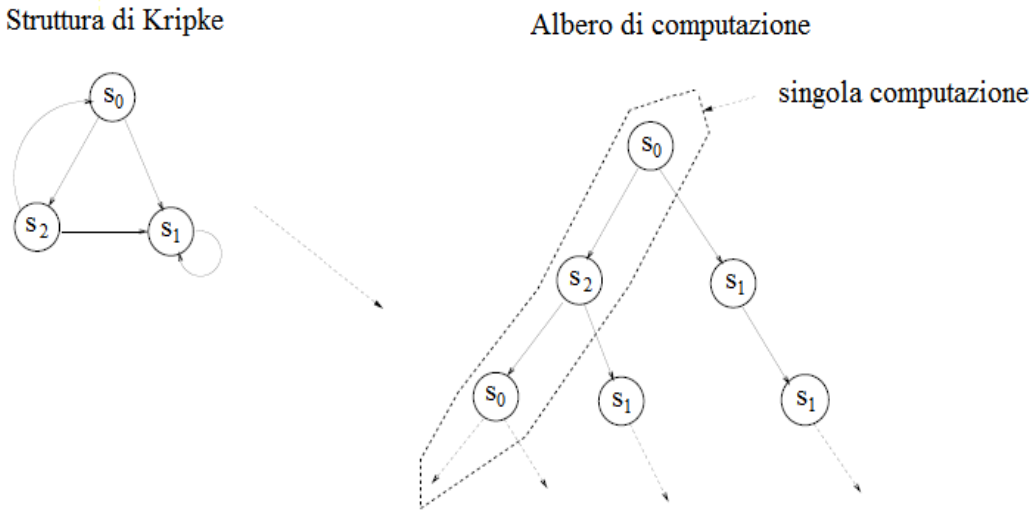

Figura 2.2.2-2

Sia φ una formula e K una struttura di kripke, allora la relazione $K,s \models \varphi$ (K soffisfa φ in s) è definita come [8]:

- $s \models p \leftrightarrow p \in L(s)$;
- $s \models \neg\varphi \leftrightarrow$ non vale $s \models \varphi$;
- $s \models \varphi \vee \psi \leftrightarrow s \models \varphi$ o $s \models \psi$;
- $s \models \varphi \wedge \psi \leftrightarrow s \models \varphi$ e $s \models \psi$;
- $s \models EX\varphi \leftrightarrow \exists s' : s' \models \varphi$ e s' è un successore di s;
- $s \models EG\varphi \leftrightarrow \exists$ una s-esecuzione : $s_i \models \varphi$ vale $\forall\ i \geq 0$;
- $s \models E(\varphi U \psi) \leftrightarrow \exists$ una s-esecuzione : per qualche $i \geq 0$ vale che $s_i \models \psi$ e $\forall\ j < i,\ s_j \models \varphi$;

2.3 Verifica

Dopo aver definito il modello e la specifica, l'ultima fase consiste nel controllare se il dato modello soddisfa o meno la data specifica. In caso di risposta positiva il ciclo finisce, altrimenti di solito viene generato un controesempio che può aiutare il modellatore a capire da dove deriva l'errore. Un ultima possibilità è quella in cui la verifica non va a buon fine a causa della dimensioni del modello.
Tutto quello che abbiamo pocanzi descritto viene illustrato nella figura seguente:

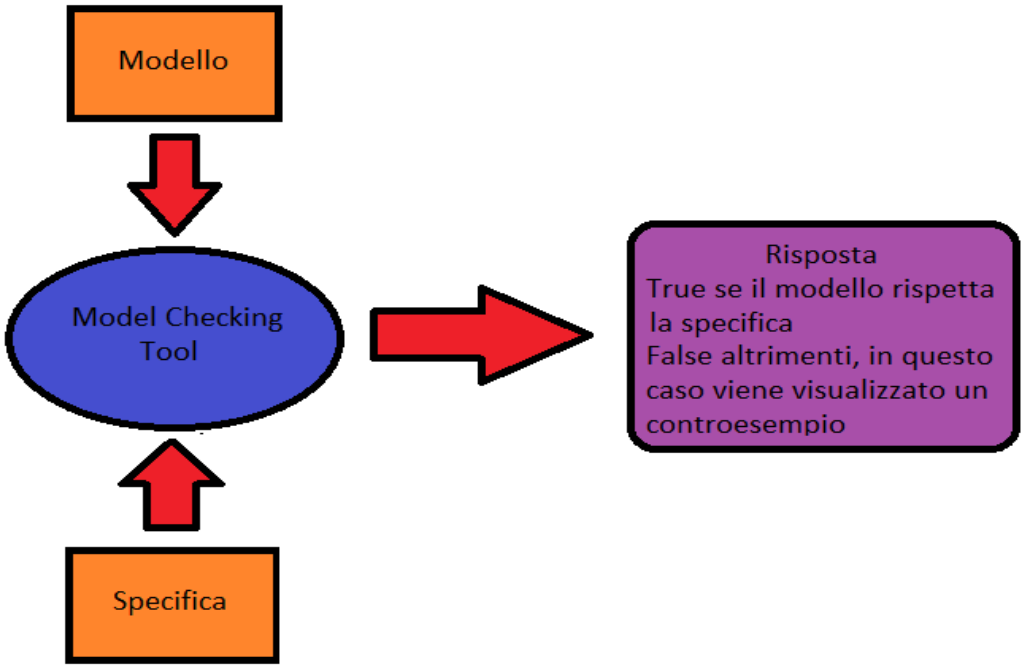

Figura 2.3-1

2.3.1 Soddisfacibilità e model checking

Data una specifica φ , il problema della soddisfacibilità [5] si basa sullo stabilire se esiste una struttura di Kripke $K = < S, S_0, R, L >$ tale che:

$$K,s \models \varphi \qquad \forall s \in S_0$$

Data una struttura di Kripke $K = < S, S_0, R, L >$ e una specifica φ , il problema del model checking [5] si basa nel verificare se per tutti gli stati $s \in S_0$ vale che:

$$K,s \models \varphi$$

Vi sono vari metodi per effettuare model checking, illustriamo di seguito quello esplicito.

2.3.2 Model checking esplicito

Questo metodo [4] etichetta ogni stato $s \in S$ con l'insieme delle sottoformule di una formula φ che sono vere in s.
Analizzeremo questo tipo di algoritmo sulla logica CTL.
Considereremo un sottoinsieme di sottoformule visto che alcuni connettori logici e operatori temporali possono essre scritti in funzione di altri.
Gli stati verrano etichettati a seconda della struttura della sottoformula considerata come segue:

- $\neg\varphi$: si etichettano con $\neg\varphi$ tutti gli stati che non sono etichettati con φ;
- $\varphi \vee \psi$: si etichettano con $\varphi \vee \psi$ tutti gli stati che sono etichettati con φ o con ψ;
- $EX\varphi$: si etichettano con $EX\varphi$ gli stati che hanno come successore uno stato etichettato da φ;
- $E(\varphi U \psi)$: bisogna trovare tutti gli stati etichettati con ψ e proseguire all'indietro rispetto a R per trovare tutti gli stati che possono essere raggiunti etichettati da φ. Tutti gli stati trovati vengono etichettati con $EX\varphi$.
- $Eg\varphi$: si construisce una struttura K' a partire da K eliminando i nodi non etichettati da φ e restringendo R e L di conseguenza. Si considera il grafo sotteso dalla struttura di Kripke K' e lo si divide in componenti

fortemente connesse non banali. Si trovano poi gli stati che appartengono alle componenti fortemente connesse costruite e proseguendo all'indietro rispetto a R si trovano tutti quegli stati che possono essere raggiunti da un cammino in cui tutti gli stati sono etichettati da φ.

Per gestire una generica formula CTL, si applicano le regole di etichettamento appena introdotte partendo con le formule più interne.
Il costo totale dell'algoritmo è O($|\varphi| \cdot (|S| + |R|)$).

2.4 Model checking basato su automi

L'idea alla base di questo approccio [1] è che per ogni formula in logica temporale possiamo costruire un automa su una struttura infinita che accetta esattamente tutte le computazioni che soddisfano la formula. Per LTL la struttura viene rappresentata come parole infinite, mentre per CTL come alberi infiniti. Inizialmente, la formula in logica temporale veniva trasformata in un automa non -deterministico, ma questo tipo di risoluzione aveva due svantaggi evidenti: primo che la trasformazione non è per niente banale e secondo che per passare dalla logica, sia essa LTL o CTL, ad un automa abbiamo un salto esponenziale.
É stato successivamente dimostrato che utilizzando gli automi alternati al posto dei non-deterministici questo due problematiche sono state risolte. Prima di vedere come viene effettuata questo tipo di verifica entriamo un po' nel dettaglio sulle caratteristiche degli automi che andremo ad utilizzare.

2.4.1 Automa di Büchi non-deterministico su parole infinite

Un automa di Büchi non-deterministico [1] è una tupla $B = (\Sigma, S, s^0, \delta, F)$, dove Σ è un alfabeto finito e non vuoto, S è un insieme finito e non vuoto degli stati, $s^0 \in S$ è uno stato iniziale, $\delta : S \times \Sigma \to 2^S$ è una funzione di transizione e $F \subseteq S$ è un insieme di stati finali o di accettazione.

L'unica caratteristica che differenzia un automa di Büchi non-deterministico da un semplice automa non-deterministico è la sua condizione di run di accettazione. Un run r di B su una parola infinita $w = w_0, w_1, w_2, \ldots$ su Σ è una sequenza $s_0, s_1, s_2, \ldots$ dove $s_0 = s^0$ e $s_{i+1} \in \delta(s_i, w_i)$, per tutti gli $i \geq 0$.
Un run r è accettante se esiste qualche stato $s \in F$ che si ripete infinitamente spesso in r. Dall'ultima affermazione deriva che una parola infinita w è accetta da B se esiste un run accettante di B su w. L'insieme delle parole accettate dall'automa B è denotato con L (B).

2.4.2 Automa di Büchi alternato su parole infinite

Dato un insieme X, sia β(X) l'insieme delle formule Booleane positive su X (cioè le formule booleane costruite su X senza la negazione) dove inseriamo anche le formule true e false. Data la seguente definizione possiamo definire un automa di Büchi alternato [1] la seguente tupla $BA = (\Sigma, S, s^0, \delta, F)$, dove la sola differenza rispetto agli automa non-deterministici visti in precedenza è la funzione di transizione δ che è così definita : $\delta : S \times \Sigma \rightarrow \beta(X)$. Questo determina che possiamo avere per esempio la seguente transizione: $\delta(s, w) = (s_1 \perp s_2) \wedge s_3$, cioè che l'automa accetta la parola aw, dove a è un simbolo e w è una parola, quando è in uno stato s se si accetta la parola w da s_1, s_3 oppure da s_2, s_3. Gli automi alternati da come abbiamo potuto vedere combinano scelte non-deterministiche (simbolo or) a scelte universali (simbolo end).
Poiché esistono scelte universali nelle transizioni, un run r di un automa alternato non è più una semplice sequenza ma piuttosto un albero.
Quindi un run r su BA su una parola infinita $w = w_0, w_1, w_2, \ldots$ è un S-labeled tree r. Un S-labeled tree è una coppia (T,V) dove T è un albero e V è una funzione di etichettatura $V : nodi(\tau) \rightarrow S$. Un albero è un insieme $T \subseteq N^*$ tale che se $x \cdot c \in T$ dove $x \in N^*$ e $c \in N$, allora anche $x \in T$ e per ogni $0 \leq c' < c$, $x \cdot c' \in T$. Gli elementi di T vengono denominati nodi e la parola vuota ε è la radice di T. Preso un nodo $x \in T$, i nodi $x \cdot c$ dove $c \in N$ vengono detti

successori di x. La funzione g(x) rappresenta il grado di un nodo x, il grado specifica quando successori ha un nodo. Se un nodo non ha successori viene detto nodo foglia. Infine un percorso π di un albero T è un insieme $\pi \subseteq T$ tale che la radice appartiene a π e per ogni $x \in \pi$ o x è una foglia oppure esiste un unico $c \in N$ tale che $x \cdot c \in \pi$.

Un run r è accettante se ogni ramo infinito include infinitamente spesso le etichette in F. Cioè, un ramo in un run accettante o ha transizioni a true oppure transita infinitamente spesso in stati di accettazione.

Prima di chiudere questa parte notiamo che gli automi di Büchi alternanti sono una generalizzazione di quelli non-deterministici, infatti i non-deterministici restringono la loro funzione β al solo utilizzo dell operatore logico or.

2.4.3 Trasformazione di una formula in LTL ad automa

Il seguente teorema stabilisce una trasformazione da una formula LTL ad un automa alternato di Büchi.

TEOREMA 1 [2]

Data una formula φ in LTL, si può costruire un automa alternato di Büchi $BA_\varphi = (\Sigma, S, s^0, \delta, F)$, dove $\Sigma = 2^{AP}$ e |S| è in O(|φ|), tale che L(BA_φ) è esattamente l'insieme delle computazioni che soddisfano la formula φ.

Dimostrazione

L'insieme degli stati di S consiste di tutte le sottoformule di φ e delle loro negazioni. Lo stato iniziale s^0 è proprio la formula φ. L'insieme F consiste di tutte le formule in S che hanno la forma $\neg(\varphi U \psi)$. Rimane da definire la funzione di transizione δ. Sapendo come rappresentare da una formula il suo duale, definiamo δ come segue:

- $\delta(p,a)$ = true se $p \in a$;
- $\delta(p,a)$ = false se $p \notin a$;

- $\delta(\varphi \wedge \psi, a) = \delta(\varphi, a) \wedge \delta(\psi, a)$;
- $\delta(\neg\varphi, a) = \delta(\varphi, a)$;
- $\delta(X\varphi, a) = \varphi$;
- $\delta(\varphi U \psi, a) = \delta(\psi, a) \vee (\delta(\varphi, a) \wedge \varphi U \psi)$;

Esempio [2].

Consideriamo la seguente formula $\varphi = (X \neg p) U q$. L'automa di Büchi alternato corrispondente è il seguente : $BA_\varphi = (2^{\{p,q\}}, \{\varphi, \neg\varphi, X\neg p, \neg X \neg p, p, \neg p, q, \neg q\}, \varphi, \delta, \neg\varphi)$, la funzione δ è descritta di seguito:

s	$\delta(s, \{p,q\})$	$\delta(s, \{p\})$	$\delta(s, \{q\})$	$\delta(s, \square)$
φ	true	$\neg p \wedge \varphi$	true	$\neg p \wedge \varphi$
$\neg\varphi$	false	$p \vee \neg\varphi$	false	$p \vee \neg\varphi$
$X \neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$
$\neg X \neg p$	p	p	p	p
p	true	true	false	false
$\neg p$	false	false	true	true
q	true	false	true	false
$\neg q$	false	true	false	true

COROLLARIO 1 [2]

Data una formula φ in LTL, si può costruire un automa non-deterministico di Büchi $B_\varphi = (\Sigma, S, s^0, \delta, F)$, dove $\Sigma = 2^{AP}$ e |S| è in $2^{O(|\varphi|)}$, tale che $L(B_\varphi)$ è esattamente l'insieme delle computazioni che soddisfano la formula φ.

2.4.4 Model checking basato su automi in LTL

Adesso abbiamo tutti gli strumenti per poter analizzare il model checking con automi sulla logica LTL [1]. L'algoritmo è il seguente:

1. Si rappresenta il modello (nel nostro caso la Kripke) con un automa non-deterministico di Büchi, dove $\Sigma = 2^{AP}$ e $t \in \delta(u, a)$ se e solo se esiste R(u,t) e a = L(u);
2. si rappresenta la proprietà con una formula LTL;
3. si considera la proprietà negata e se ne costruisce l'equivalente automa non-deterministico di Büchi;
4. si intersecano i linguaggi accettati dai due automi;
5. si verifica se l'intersezione è vuota;

Questo algoritmo si basa su un semplice ragionamento: il linguaggio riconosciuto dall'automa del modello deve essere interamente contenuto nel linguaggio riconosciuto dall'automa della formula, quindi questo problema è equivalente a verificare il vuoto dell'intersezione tra l'automa del modello e l'automa della formula negata.

TEOREMA 2 [1]
Verificare che un struttura di Kripke K soddisfa una formula LTL φ può essere fatta in tempo $O(|K| * 2^{O(|\varphi|)})$ e in spazio $O((|\varphi| + \log|K|)^2)$.

Come per LTL vi è un algoritmo di model checking anche per CTL. In questo caso, come accennato in precedenza, gli automi che analizzeremo non saranno più su parole infinite ma su alberi infiniti.

2.4.5 Automa non-deterministico su alberi binari infiniti

Un automa non-deterministico su alberi binari infiniti [3] può essere descritto con la tupla $Tb = (\Sigma, S, s^0, \delta, F)$, dove Σ è un alfabeto, S è l'insieme finito degli stati, s^0 è lo stato iniziale, F è l'insieme degli stati di accettazione e $\delta : S \times \Sigma \to 2^{S^2}$ è la funzione di transizione che mappa uno stato $s \in S$ e una lettera di input $\sigma \in \Sigma$ con un insieme di coppie di stati. Ogni coppia rappresenta

una scelta, prendiamo un qualsiasi stato s e immagginiamo che legga un nodo x etichettato con la lettera σ, ipotizziamo proceda con la scelta della coppia $(s_1, s_2) \in \delta(s, \sigma)$, ora l'automa si divide in due copie. Una copia entra nello stato s_1 e procede per il figlio sinistro di x, e l'altra copia entra nello stato s_2 e procede per il figlio destro di x.

2.4.6 Automa alternato su alberi infiniti

La funzione di transizione dell'automa non-deterministico su alberi binari possiamo anche vederla nel seguente modo:

$$\delta(s, \sigma) = \{(s_1, s_2), (s_2, s_4)\} = (0, s_1) \wedge (1, s_2) \vee (0, s_2) \wedge (1, s_4)$$

Nel automa non-deterministico su alberi bisogna ogni and nella funzione di transizione ha esattamente un elemento associato con ogni direzione. Negli automi alternati su alberi binari invece la funzione di transizione può essere un arbitraria formula. Ecco un esempio: $\delta(s, \sigma) = (0, s_1) \wedge (0, s_2) \vee (0, s_2) \wedge (1, s_4) \wedge (1, s_3)$. Qui si nota che vi possono essere più copie che vanno nella stessa direzione ed allo stesso tempo che l'automa non richiede che per ogni direzione vi sia almeno una copia.

A livello formale un automa alternato su alberi binari [3] è una tupla ATb = (Σ, S, s^0 , δ, F) dove Σ è un alfabeto, S è l'insieme finito degli stati, s^0 è lo stato iniziale, F è l'insieme degli stati di accettazione e $\delta : S \times \Sigma \to \beta(\{0,1\} \times S)$.

Per adesso abbiamo parlato solo di automi su alberi binari, adesso generalizziamo al caso in cui i nodi possono avere grado maggiore di 2. L'unica restrizione che effettuiamo è che l'insieme di tutti i possibili gradi è finito e ne conosciamo la dimensione massima. In altre parole, noi richiediamo che l'automa abbia un altro elemento nella tupla $D \subset N$ che rappresenti il grado massimo di qualsiasi nodo.

Ora possiamo definire formalmente un automa alternato su alberi infiniti [3] che rappresenta la seguente tupla $AT = (\Sigma, D, S, s^0, \delta, F)$ e la sua funzione di transizione è così definita $\delta : S \times \Sigma \times D \to \beta(N \times S)$ con il requisito che per ogni $k \in D$ abbiamo che $\delta(s, \sigma, k) \in \beta(\{0 \ldots k-1\} \times S)$.

Un run di un automa alternato su un albero (T,V) è un albero (T_r, r) nel quale la radice è etichettata con s_0 e tutti gli altri nodi sono etichettati con elementi di $N^{\perp} x S$. Ogni nodo di T_r corrisponde ad un nodo di T e per ogni nodo di T possono corrispondere molti nodi di T_r. Notiamo che questo va in contrasto, come era facile da dedurre, rispetto alla della definizione dell'automa non-deterministico su alberi in cui ogni nodo di T corrisponde ad uno ed un solo nodo dell'albero del run.

Un run (T_r, r) è accettante se tutti i suoi percorsi infiniti soddisfano la condizione di accettazione. Quindi nel caso specifico di automi di Büchi ogni percorso infinito deve contenere infinitamente spesso almeno uno stato che appartiene ad F.

2.4.7 Model checking basato su automi in CTL

Potremmo chiederci perchè vi sono due approcci differenti di model checking basato su automi in base alla logica che stiamo utilizzando per rappresentare la formula. La risposta sta nella rappresentazione delle computazioni su una struttura di Kripke. In LTL ogni struttura di Kripke può essere rappresentata da infinite computazioni, mentre in CTL ogni struttura di Kripke è rappresentata da una singola computazione non-deterministica [3]. Quindi in CTL il model checking consiste nel valutare se la singola computazione appartiene al linguaggio riconosciuto dall'automa della formula.

Usiamo anche qui automi alternanti perchè riducono la grandezza dell'automa da esponenziale sulla lunghezza della formula a lineare [3].

Adesso cerchiamo di analizzare nel dettaglio tutto il processo [3].

Notiamo che una Kripke $K = < S, s_0, R, L >$ può essere vista come un albero (T_K, V_K) che corrisponde allo srolotamento di K da s_0. Per ogni nodo s, definiamo g(s) come il grado di s (cioè il numero di successori che ha s) e con $succ(s) = (s_0, s_1, ..., s_{g(s)-1})$ la lista ordinata di successori di s.

Definiamo adesso T_K, V_K induttivamente come segue:

1. $V_K(\varepsilon)=s_0$;
2. Per $y \in T_K$ con $succ(V_K(y)) = (s_0, s_1, ..., s_m)$ e per $0 \leq i \leq m$, abbiamo $y \cdot i \in T_K$ e $V_K(y \cdot i)=s_i$;

Sia φ una formula in CTL e sia $D \subset N$ l'insieme dei gradi di una struttura di Kripke K. Definiamo l'automa alternato $AT_{D,\varphi}$ che accetta esattamente tutti i D-alberi che soddisfano φ. Consideriamo a questo punto il prodotto di K con $AT_{D,\varphi}$, cioè un automa che accetterà il seguente linguaggio:

$$L(AT_{D,\varphi}) \cap \{(T_K, V_K)\}$$

Come è facile notare il linguaggio di questo prodotto o conterrà il singolo albero (T_K, V_K) , nel qual caso K rispetta la specifica φ, oppure sarà vuoto e quindi K non rispetterà la specifica.

L'algoritmo su cui si basa il model checking è il seguente:

1. Costruisco l'automa alternato della formula con grado massimo D $AT_{D,\varphi}$;
2. Costruisco l'automa alternato $AT_{K,\varphi} = K \times AT_{D,\varphi}$;
3. La risposta sarà true se il linguaggo di $AT_{K,\varphi}$ è diverso dal vuoto, false altrimenti.

Nel prossimo capitolo tratteremo la difficoltà più grande riguardante il model checking, cioè l'esplosione degli stati, e valuteremo alcuni metodi per limitarne il problema.

3 Il problema dell'esplosione degli stati

Nel capitolo precendente abbiamo illustrato come effettuare il processo di model checking soprattutto dal punto di vista teorico sviluppando principalmente quello basato sugli automi. Ora cerchiamo di analizzare a livello pratico quali problematiche vengono a presentarsi. Il principale problema del model checking è l'esplosione degli stati [7] dovuta dal fatto che un modello rappresenta lo spazio degli stati del sistema e quindi la sua dimensione è esponenziale nella dimensione della descrizione del sistema. Questo porta che anche in sistemi relativamente piccoli, molte volte è impossibile computare il loro modello. Per risolvere questo delicato problema sono stati sviluppati vari metodi, quelli che analizzeremo sono:

- Model checking simbolico [8] è un metodo di verifica in cui la relazione di transizione del modello non è costruita esplicitamente, ma viene calcolata una funzione booleana che rappresenta tale funzione di transizione;
- Model checking astratto [7] è un metodo di verifica che sfrutta la conoscenza sul modello e sulla specifica in modo da modellare solo le caratteristiche rilevanti;
- Model checking gerarchico [11] è un metodo di verifica in cui il modello è rappresentato da stati e super-stati (o blocchi), con la caratteristica che ogni super-stato rappresenta un altro modello.

Dal prossimo paragrafo analizzeremo nel dettaglio questi tipi di approcci.

3.1 Model checking simbolico

In questo tipo di verifica sia la relazione di transizione che gli insiemi di stati sono rappresentati tramite funzioni booleane. In molte situazioni pratiche questo tipo di approccio è esponenzialmente più piccolo della rappresentazione esplicita, quindi

rappresentare tutto come funzioni booleane diminuisce il problema dell'esplosione degli stati. Una formula booleana può essere rappresentata tramite un albero binario di decisione.

3.1.1 Alberi binari di decisione

Un albero binario di decisione [17] è rappresentato da due tipi di nodi:

- i nodi non terminali, rappresentati graficamente con nodi tondi, sono etichettati da proposizioni atomiche e hanno sempre due archi uscenti etichettati uno a true e l'altro a false;
- i nodi terminali, rappresentati graficamente con nodi quadrati, sono etichettati con true o false.

Prendiamo ad esempio [17] la seguente formula: f(x,y) = $\neg(x \vee y)$, il suo albero binario di decisione è il seguente:

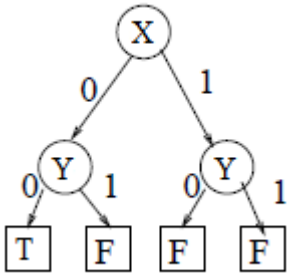

Figura 3.1.1-1

Gli alberi binari di decisione hanno dimensione esponenziale nella dimensione della formula. Per ridurre la rappresentazione dobbiamo eliminare le ridondanze, per fare questo utilizziamo le seguenti due regole [8]:

- combiniamo tutti i sottoalberi isomorfi;
- tutti i nodi con figli isomorfi vengono eliminati.

Così otteniamo un D.A.G (grafo diretto aciclico), che viene chiamato BDD (Binary Decision Diagram).

3.1.2 Binary Decision Diagram

Come detto nel paragrafo procedente questo tipo di struttura [8] ottimizza gli alberi binari di decisione, ora cerchiamo di analizzare nel dettaglio, tramite un esempio, se questa è veramente la struttura che fa al caso nostro.

Esempio.

llustriamo questo processo prendendo la seguente formula:

$$f(a_{1,}a_{2,}b_{1,}b_{2})=(a_1\leftrightarrow b_1)\wedge(a_2\leftrightarrow b_2)$$

Nella seguente figura viene illustrato l'albero binario di decisione con tutte le sue ridondanze:

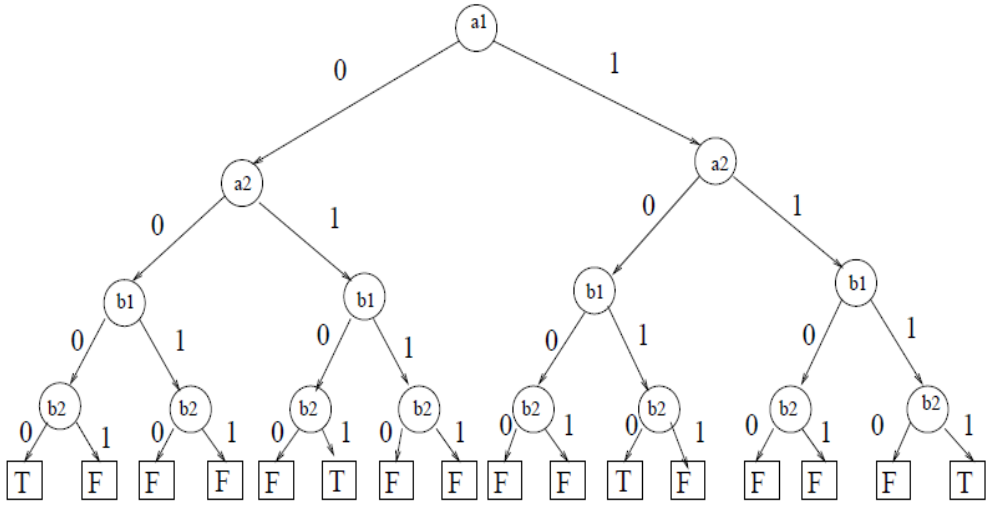

Figura 3.1.2-1

Ed ecco come può essere rappresenta la formula sfruttando i BDD:

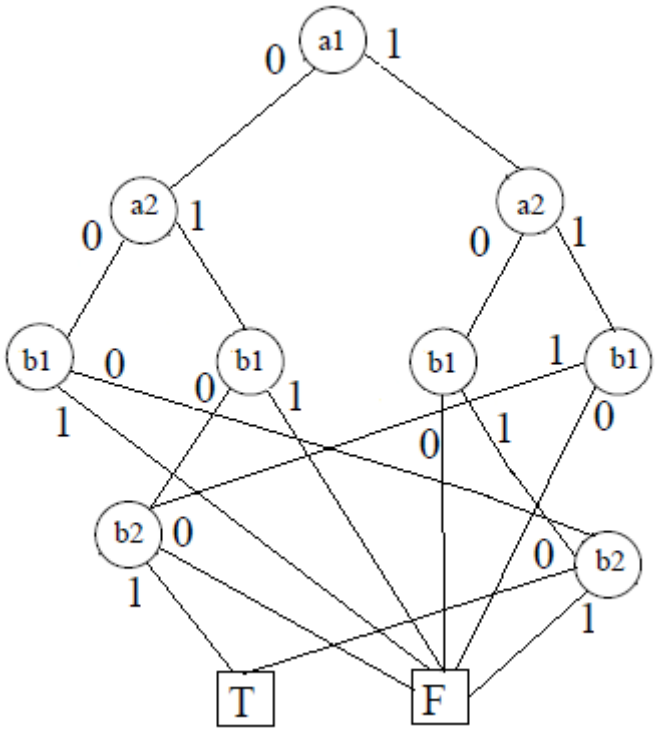

Figura 3.1.2-2

Notiamo che con i BDD riduciamo di molto il numero di nodi, ma che se ordinavamo le variabili in modo differente avremmo prodotto sicuramente un risultato migliore. Questo problema verrà analizzato nel successivo paragrafo.

3.1.3 Ordered Binary Decision Diagrams

L'OBDD [8] è simile ai BDD solo che qui le variabili vengono ordinate in modo che:

- in ogni cammino dalla radice alle foglie rispettiamo lo stesso ordinamento, senza ripetizioni;
- non tutte le variabili devono comparire lungo un cammino;

- non devono esistere sottoalberi isomorfi o nodi ridondanti nel diagramma.

Dopo aver ordinato le variabili, per ottenere un OBBD bisogna impiegare le seguenti regole [8] fin quando è possibile:

- eliminare tutti i nodi terminali lasciandone solamente uno true ed un altro false;
- se due nodi non terminali u e v hanno la stessa variabili e gli stessi figli, eliminarne uno dei due e ridirezionare i suoi archi entranti nell'altro nodo;
- se un nodo non terminale u ha due figli uguali, eliminare u e ridirezionare i nodi entranti all'unico figlio.

Questo algoritmo si può implementare tramite una visita dalle foglie alla radice.

Rivalutiamo l'esempio [8] precentemente esposto sui BDD per analizzare i miglioramenti che otteniamo.

Mostriamo per prima cosa l'albero binario di decisione con le variabili ordinate nel seguente modo:

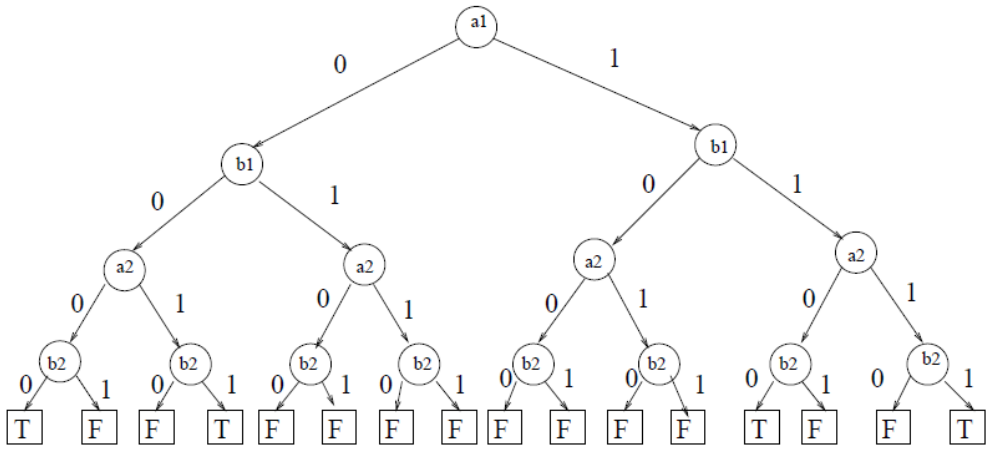

Figura 3.1.3-1

Ed ora vengono illustrate tutte le ottimizzazioni, iniziando dall'eliminazione di tutte le foglie ridondanti:

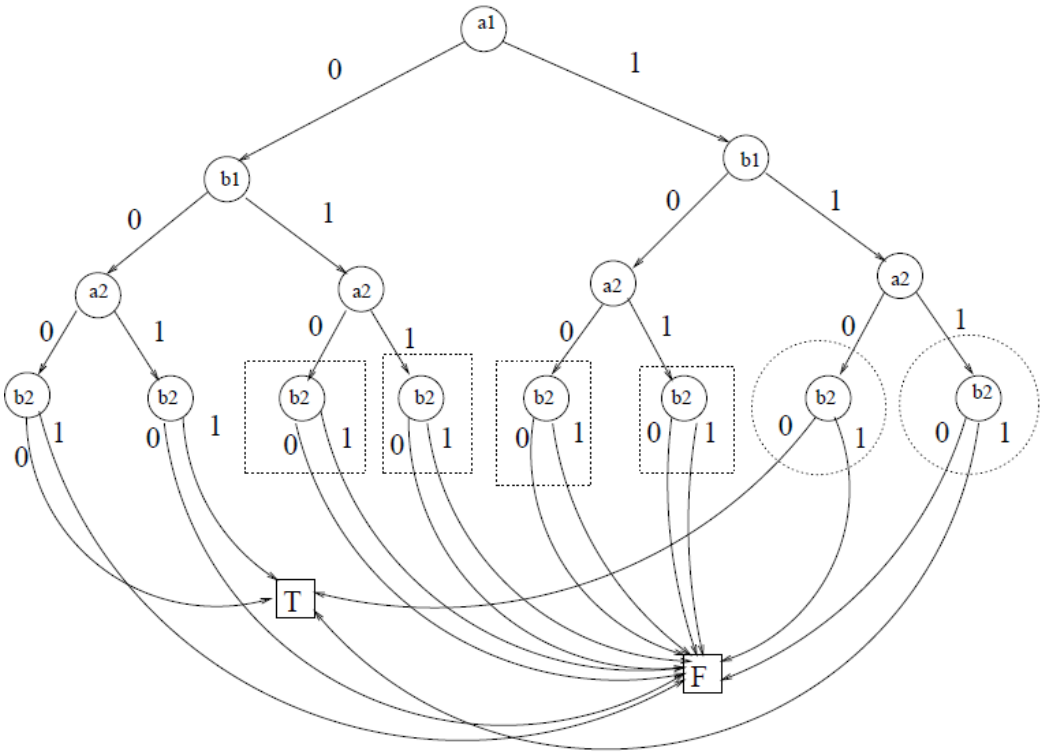

Figura 3.1.3-2

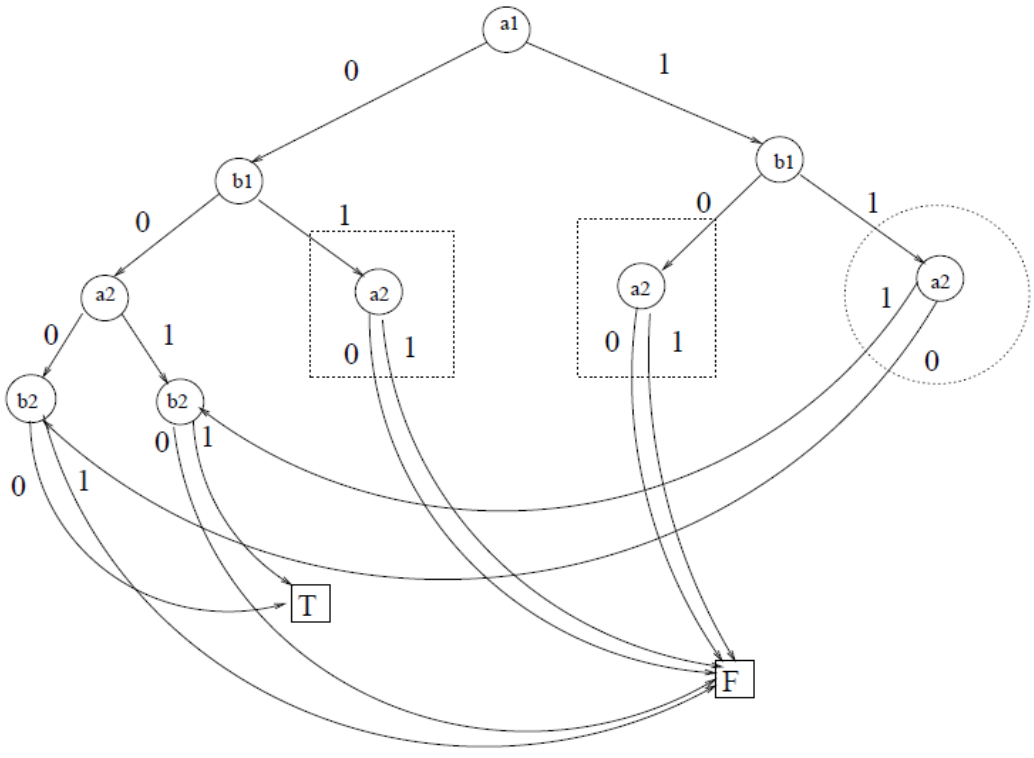

Figura 3.1.3-3

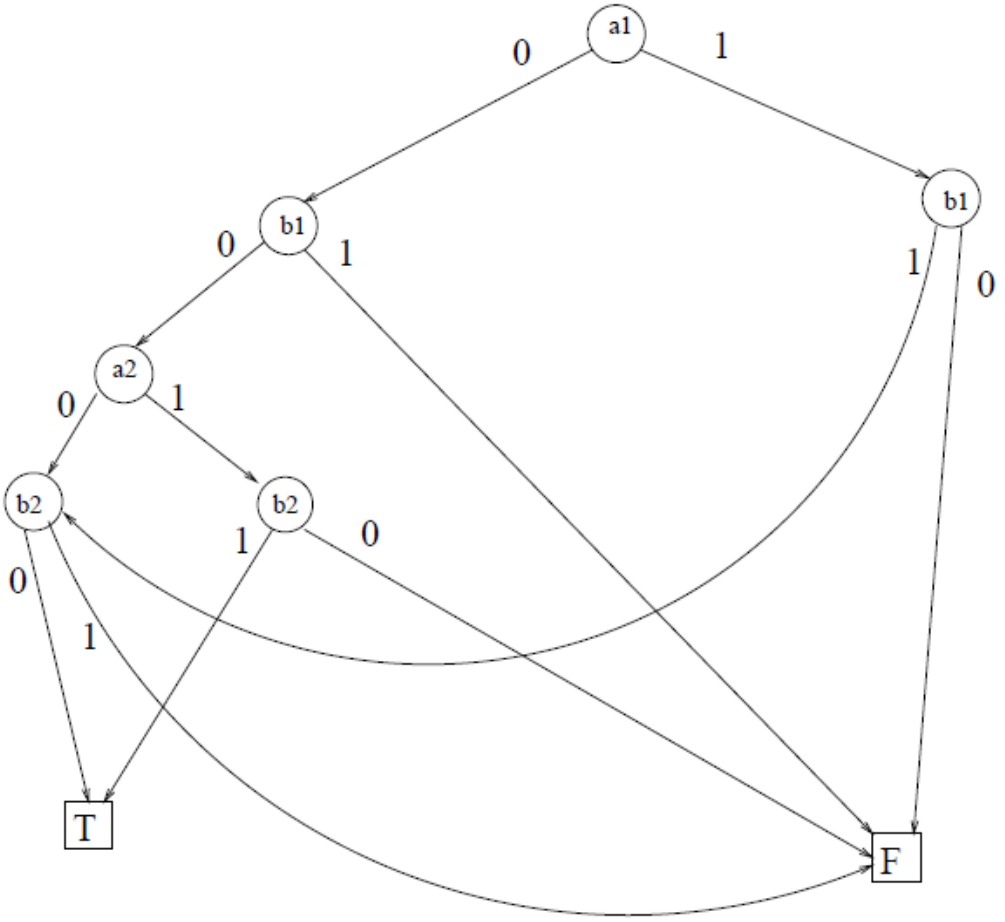

Figura 3.1.3-4

3.1.4 Rappresentare Strutture di Kripke

Per rappresentare una struttura di Kripke K = $< S, S_0, R, L >$ in un OBDD [8] prima dobbiamo rappresentare K in termini di formule booleane. Per fare questo utilizzeremo come rappresentazione simbolica le disgiunzioni di congiunzioni (DNF). Definito $AP = \{p_1, \ldots, p_n\}$ l'insieme di proposizioni atomiche, possiamo rappresentare ogni stato $s \in S$ con la funzione booleana:

$$f_s = \bigwedge_{i=1}^{|AP|} l_i$$

dove $l_i = p_i$ se $p_i \in L(s)$ e $l_i = \neg p_i$ altrimenti.

Definito un singolo stato è molto semplice rappresentare in formula booleana un insieme finito di stati $s_1, \ldots, s_k$:

$$\bigvee_{i=1}^{k} (l_{i1} \wedge \ldots \wedge l_{in})$$

dove $l_{ij} = p_{ij}$ se $p_{ij} \in L(s_i)$ e $l_{ij} = \neg p_{ij}$ altrimenti.

Ora non ci resta che rappresentare le transizioni, per fare questo introduciamo i simboli $p'_1, \ldots, p'_n$ che rappresentano i valori delle proposizioni atomiche sullo stato raggiunto. Quindi un arco da s a s' si rappresenta con la formula:

$$l_1 \wedge \ldots \wedge l_n \wedge l'_1 \wedge \ldots \wedge l'_n$$

dove $l_i = p_i$ se $p_i \in L(s)$ e $l_i = \neg p_i$ altrimenti, ed $l'_i = p_i$ se $p_i \in L(s')$ e $l'_i = \neg p_i$ altrimenti.

Come fatto per gli stati, adesso siamo in grado di rappresentare l'insieme delle transizioni della struttura di Kripke :

$$\bigvee_{i=1}^{k} (l_{i1} \wedge \ldots \wedge l_{in} \wedge l'_{i1} \wedge \ldots \wedge l'_{in})$$

dove $l_{ij} = p_{ij}$ se $p_{ij} \in L(s_i)$ e $l_{ij} = \neg p_{ij}$ altrimenti, ed $l'_{ij} = p_{ij}$ se $p_{ij} \in L(s'_i)$ e $l'_{ij} = \neg p_{ij}$ altrimenti.

3.2 Model checking astratto

Questo metodo [10] risolve il problema dell'esplosione degli stati rimpiazzando una Struttura di Kripke complessa con una più piccola che soddisfa le stesse proprietà di interesse. Per garantire le stesse proprietà noi necessitiamo di introdurre 2 funzioni di transizione, una che preservi le proprietà universali e un'altra le proprietà esistenziali. Queste esigenze vengono soddisfatte da una Kripke Modal Transition System (KMTS) [10].
Una KTMS è una tupla $K_a = < S, S_0, \rightarrow_{must}, \rightarrow_{may}, L >$, dove:

1. S è un insieme finito di stati;
2. $S_0 \subseteq S$ è un insieme di stati iniziali;
3. $\rightarrow_{must} \subseteq S \times S$ e $\rightarrow_{may} \subseteq S \times S$ sono funzioni di transizione tali che la relazione $\rightarrow_{may}$ è totale e $\rightarrow_{must} \subseteq \rightarrow_{may}$;
4. $L : S \rightarrow 2^{Lit}$ è una funzione di etichettatura che associa ogni stato in S con litterali di Lit, tale che per ogni stato s e proposizione atomica $p \in AP$, al massimo uno tra p e $\neg p$ è in L(s).

Una transizione $\rightarrow_{must}$ viene chiamata must-transition e allo stesso modo una transizione $\rightarrow_{may}$ viene chiamata may-transition. Un must-path (may-path) in K_a è una sequenza massimale di stati $\pi = s_0, s_1, ...$ tale che per ogni due stati consecutivi s_i, s_{i+1} in π, noi abbiamo che $s_i \rightarrow_{must} s_{i+1}$ ($s_i \rightarrow_{may} s_{i+1}$). Una sequenza di stati viene detta massimale quando non può essere estesa da qualsiasi altra transizione dello stesso tipo. Visto che la funzione $\rightarrow_{must}$ non è totale, e quindi un percorso può anche essere finito, deduciamo che un must-path non è per forza anche un may-path come invece accade per le singole transizioni.
Dopo aver definito una Kripke Modal Transition System è facile notare che una struttura di Kripke può essere vista come una KTMS dove $\rightarrow = \rightarrow_{must} = \rightarrow_{may}$ e per ogni s e proposizione atomica $p \in AP$, abbiamo esattamente p o $\neg p$ in

L(s) [10].

Noi consideriamo un tipo di astrazione dove un insieme di stati concreti in una struttura di Kripke K_c vengono collassati in un singolo stato astratto in una KTMS K_a . Ora possiamo illustrare come da un modello concreto viene prodotto un modello astratto [10].

Definizione 3.2.1 [10]

Siano $(C, *)$ e $(A, \star)$ due insiemi parzialmente ordinati.

Allora $\alpha : C \to A, \gamma : A \to C$ è una connessione di Galois da $(C, *)$ a $(A, \star)$ se e solo se:

1. α e γ sono funzioni totali e monotone;
2. per ogni $c \in C$ vale $\gamma \circ \alpha(c) * c$;
3. per ogni $a \in A$ vale $\alpha \circ \gamma(a) \star a$.

Sia $K_c = < S_c, S_{0c}, \to, L_c >$ una struttura di Kripke concreta.

Sia $(S_a, \star)$ un insieme parzialmente ordinato di stati astratti e sia $(\gamma : S_a \to 2^{S_c}, \alpha : 2^{S_c} \to S_a)$ una connessione di Galois da $(2^{S_c}, *)$ a $(S_a, \star)$. Allora γ è la funzione che mappa ogni stato astratto con l'insieme degli stati concreti e α è la funzione che associa ad ogni insieme di stati concreti lo stato astratto che li rappresenta.

A questo punto possiamo definire un modello astratto $K_a = < S_a, S_{0a}, \to_{must}, \to_{may}, L_a >$ come segue [10]:

- l'insieme degli stati iniziali S_{0a} è costruito associando ad ogni stato iniziale concreto uno ed un solo stato iniziale astratto, senza alcuna aggiunta;
- una must-transition $s_a \to_{must} s_a'$ esiste solo se $\forall s_c \in \gamma(s_a)$ abbiamo che $\exists s_c' \in \gamma(s_a')$ tale che $s_c \to s_c'$, ricordiamo che questa funzione di

transizione può anche essere vuota;

- una may-transition $s_a \rightarrow_{may} s_a'$ esiste solo se $\exists s_c \in \gamma(s_a)$ e $\exists s_c' \in \gamma(s_a')$ tale che $s_c \rightarrow s_c'$;
- la funzione L_a di etichettatura associa ad uno stato astratto s_a l'etichetta $l \in Lit$ se e solo se tutti gli stati concreti rappresentati da s_a sono etichettati da l.

Definizione 3.2.2 [10]

Sia $K_c = < S_c, S_{0c}, \rightarrow, L_c >$ una struttura di Kripke concreta, e sia $K_a = < S_a, S_{0a}, \rightarrow_{must}, \rightarrow_{may}, L_a >$ una KMTS astratta.

Noi diciamo che $H \subseteq (S_c \times S_a)$ è una mixed-simulation da K_c a K_a se $(s_c, s_a) \in H$ implica che:

- $L_a(s_a) \subseteq L_c(s_c)$;
- se $s_c \rightarrow s_c'$ allora c'è qualche $s_a' \in S_a$ tale che $s_a \rightarrow_{may} s_a'$ e $(s_c', s_a') \in H$;
- se $s_a \rightarrow_{must} s_a'$ allora c'è qualche $s_c' \in S_c$ tale che $s_c \rightarrow s_c'$ e $(s_c', s_a') \in H$.

Se esiste una mixed-simulation H tale che per ogni $s_c \in S_{0c}$ esiste $s_a \in S_{0a}$ tale che $(s_c, s_a) \in H$ e per ogni $s_a \in S_{0a}$ esiste $s_c \in S_{0c}$ tale che $(s_c, s_a) \in H$ diciamo che K_a è più astratta di K_c .

Ora definiamo la semantica a 3 valori per una formula CTL su una KMTS.

Una semantica a 3 valori preserva sia la soddisfazione (tt) che la confutazione (ff) di una formula dal modello astratto ad uno concreto, ed inserisce un nuovo valore di verità $\perp$. Se il valore di verità di una formula in un modello astratto è $\perp$ significa che il valore di verità nel modello concreto non è conosciuto e quindi può essere o tt oppure ff.

Definizione 3.2.3 [10]

La semantica a 3 valori di una formula CTL φ in uno stato s di una KMTS K_a =

$< S, S_0, \rightarrow_{must}, \rightarrow_{may}, L >$ denotatata con $s \models_3 \varphi$ è definita induttivamente come segue:

- $s \models_3 true = tt$;
- $s \models_3 false = ff$;
- $s \models_3 l =$
  - tt se $l \in L(s)$;
  - ff se $\neg l \in L(s)$;
  - $\perp$ altrimenti;
- $s \models_3 \psi \wedge \varphi =$
  - tt se $s \models_3 \psi = tt$ e $s \models_3 \varphi = tt$;
  - ff se $s \models_3 \psi = ff$ o $s \models_3 \varphi = ff$;
  - $\perp$ altrimenti;
- $s \models_3 \psi \vee \varphi =$
  - tt se $s \models_3 \psi = tt$ o $s \models_3 \varphi = tt$;
  - ff se $s \models_3 \psi = ff$ e $s \models_3 \varphi = ff$;
  - $\perp$ altrimenti;
- $s \models_3 A\varphi =$
  - tt se per ogni may-path π da s: $\pi \models_3 \varphi = tt$;
  - ff se esiste un must-path π da s: $\pi \models_3 \varphi = ff$;
  - $\perp$ altrimenti;
- $s \models_3 E\varphi =$
  - tt se esiste un must-path π da s: $\pi \models_3 \varphi = tt$;
  - ff se per ogni may-path π da s: $\pi \models_3 \varphi = ff$;
  - $\perp$ altrimenti;

Per un may o must path $\pi = s_0, s_1, \ldots$, $\pi \models_3 \varphi$ è definita come segue:

- $\pi \models_3 X\varphi =$
  - $s_1 \models_3 \varphi$ se $|\pi| > 1$;
  - $\perp$ altrimenti;
- $\pi \models_3 \varphi U \psi =$
  - tt se $\exists\ 0 \leq k < \pi$:
    $[(\ s_k \models_3 \psi = tt) \ \wedge \ (\ \forall\ j < k :\ s_j \models_3 \varphi = tt\ )]$;
  - ff se $\{\ \forall\ 0 \leq k < \pi : [\ \forall\ j < k:\ s_j \models_3 \varphi \neq ff$
    $\Rightarrow \quad s_k \models_3 \psi = ff\ ]\ \} \ \wedge \ (\ \forall\ 0 \leq k < \pi :$
    $s_k \models_3 \varphi \neq ff \Rightarrow |\pi| = \infty\ )$;
  - $\perp$ altrimenti;

Noi diciamo che $K_a \models_3 \varphi = tt$ se $\forall s_0 \in S_0 : s_0 \models_3 \varphi = tt$.
Noi diciamo che $K_a \models_3 \varphi = ff$ se $\exists s_0 \in S_0 : s_0 \models_3 \varphi = ff$.
Se non vale nessuna delle due enunciazioni precedenti allora $K_a \models_3 \varphi = \perp$.
Il prossimo teorema garantisce la preservazione di una formula CTL da un modello astratto ad un modello concreto.

Teorema 3.2.1 [10]
Sia $H \subseteq (S_c \times S_a)$ una relazione di mixed-simulation da una struttura di Kripke K_c ad una Kripke Modal Transition System K_a , allora per ogni $(s_c, s_a) \in H$ e per ogni formula CTL φ, abbiamo che:

- $s_a \models_3 \varphi = tt \Rightarrow s_c \models_3 \varphi = tt$;
- $s_a \models_3 \varphi = ff \Rightarrow s_c \models_3 \varphi = ff$;

Se K_a è più astratta di K_c , allora per ogni formula CTL φ, abbiamo che:

- $K_a \models_3 \varphi = tt \Rightarrow K_c \models_3 \varphi = tt$;
- $K_a \models_3 \varphi = ff \Rightarrow K_c \models_3 \varphi = ff$;

Abbiamo visto quali relazioni intercorrono tra un modello astratto ed un modello concreto e come creare un modello astratto da uno concreto, a questo punto illustriamo, con l'aiuto della figura 3.2.1, quali sono i passi principali nel model checking astratto [7].

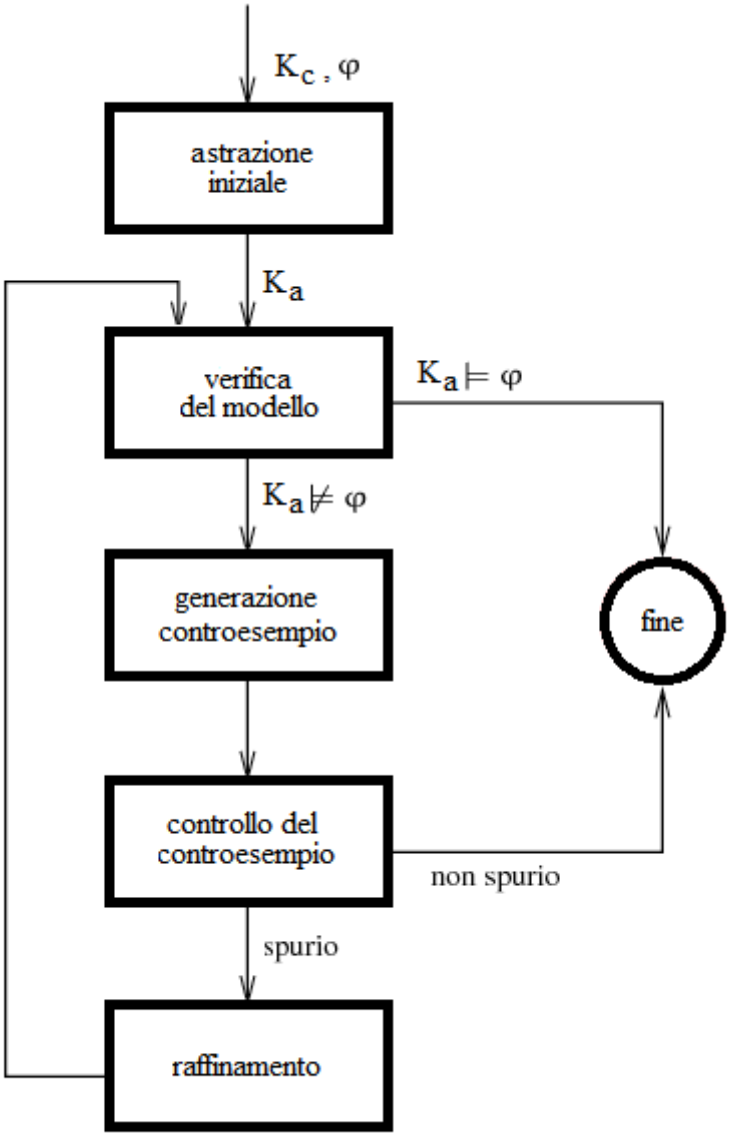

Figura 3.2.1

I passi principali sono i seguenti:

- astrazione iniziale, in questa fase prendiamo il modello concreto K_c e generiamo un astrazione K_a ;
- verifica del modello, in questa fase vediamo se la data specifica φ è verificata nel modello K_a , in caso affermativo abbiamo finito;
- generazione controesempio, in questa fase viene generato un controesempio;
- controllo del controesempio, qui bisogna valutare se il controesempio astratto corrisponde ad uno concreto (controesempio non spurio), in caso affermativo abbiamo finito;
- raffinamento, ci si trova in questa fase quando è stato generato un controesempio spurio, qui si effettua un raffinamento dell'intero modello in modo tale che la nuova struttura astratta non ammetta nuovamente il suddetto controesempio spurio.

3.3 Model checking gerarchico

Il model checking gerarchico [11] introduce come concetto di rappresentazione del modello la gerarchica. Un modello gerarchico ha la caratteristica principale di avere dei nodi che possono rappresentare altri modelli. Un esempio di modello gerarchico può essere un orologio digitale, il top-level della macchina è rappresentato da un ciclo di 24 super-stati, uno per ogni ora del giorno. Ognuno di questi super-stati è un modello gerarchico che consiste in un ciclo contentente 60 super-stati che contano i minuti, ognuno dei quali è un ordinario modello che consiste in un ciclo che conta i secondi. In base all'esempio descritto notiamo due vantaggi principali dei modelli gerarchici rispetto a quelli ordinari. In primo luogo i super-stati offrono un tipo di struttura che permette di specificare i sistemi in raffinamenti successivi e di visualizzare i differenti livelli di granularità. Il secondo vantaggio è che permette la condivisione delle componenti dei modelli

(per esempio, i 24 super-stati che rappresentano al top-level l'orologio digitale sono mappate dallo stesso modello gerarchico che rappresenta le ore), in questo modo noi specifichiamo delle componenti solo una volta e potremo riutilizzarle in diversi contesti, portando modularità e succentezza nei sistemi rappresentati.
Dato un modello gerarchico noi possiamo ottenere un modello ordinario utilizzando un algoritmo di flattening. Questo algoritmo non farà altro che sostituire ricorsivamente ogni super-stato con il modello che lo rappresenta. Tale algoritmo di flattening può causare un blow-up considerevole soprattutto quando vi è molta condivisione. Prendendo in esempio l'orologio digitale che è decritto con il modello gerarchico da 144 vertici (24+60+60) notiamo che appiattendolo avremo addirittura 86.400 (24*60*60) vertici! Quindi se noi effettuaimo il flattening prima del model checking la complessità nel caso peggiore sarà esponenziale nella descrizione originale della struttura.
Fatte queste premesse ora possiamo vedere come rappresentare formalmente una struttura di Kripke gerarchica.

3.3.1 Struttura di Kripke gerarchica

Una struttura di Kripke gerarchica K [11] su un insieme AP di proposizioni atomiche è una tupla $< K_{1,} K_{2,...}, K_n >$ di strutture di Kripke, dove ogni K_i è una tupla $< N_i, B_i, n_0^i, O_i, X_i, Y_i, E_i >$, dove:

1. N_i è l'insieme finito di nodi;
2. B_i è l'insieme finito di box;
3. $n_0^i \in N_i$ è uno stato iniziale;
4. $O_i \subset N_i$ è un insieme chiamato di exit-nodes;
5. $X_i : N_i \rightarrow 2^{AP}$ è una funzione di etichettatura che associa ad ogni nodo un sottoinsieme di AP;
6. $Y_i : B_i \rightarrow \{i+1...n\}$ è una funzione di indicizzazione che mappa ogni box dell'i-esima struttura con un indice maggiore di i. Cioè se $Y_i(b) = j$, per

un box b di una struttura K_i, allora b può essere visto come un riferimento alla definizione della struttura K_j;

7. E_i è l'insieme degli archi.

L'insieme N_i e B_i sono a due a due disgiunti.
Ogni arco in E_i è una coppia (u,v), dove:

- la sorgente u può essere un nodo di K_i oppure è una coppia (w_1, w_2) dove w_1 è un box di K_i con $Y_i(w_1)=j$ e w_2 è un exit-node di K_j;
- v può essere un nodo oppure un box di K_i.

Un esempio di struttura di Kripke gerarchica [11] è illustrato nella figura 3.3.1-1.

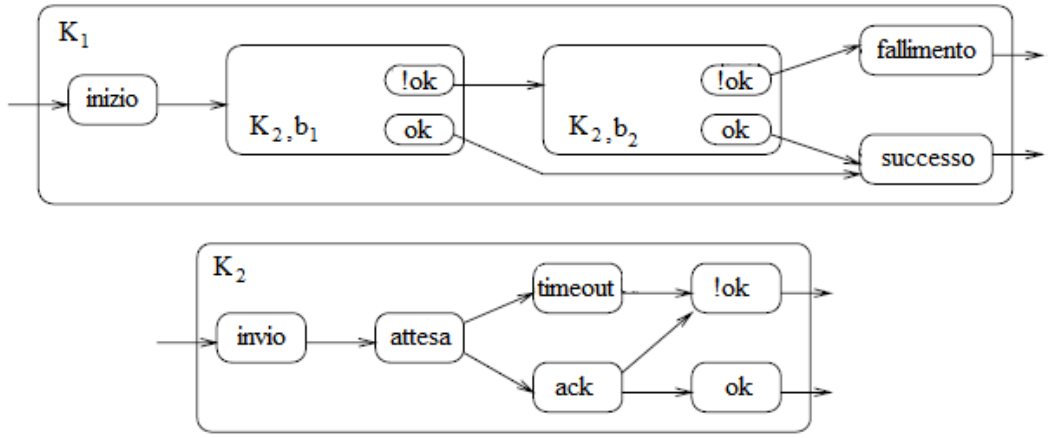

Figura 3.3.1-1

Al top-level abbiamo una struttura K_1 che ha due box, b_1 e b_2, i quali vengono entrambi mappati da K_2. La struttura K_2 rappresenta un tentativo di inviare un messaggio, il tentativo fallisce se il tempo a disposizione scade prima della ricezione dell'acknowledgment oppure se l'acknowledgment è inconsistente

rispetto al messaggio inviato. Nella struttura K_1 se il primo tentativo fallisce viene effettuato un nuovo tentativo.

Per qualsiasi struttura di Kripke gerarchica noi possiamo costruire un'ordinaria struttura di Kripke sostituendo ricorsivamente ogni box con la struttura di Kripke associata [11]. Visto che differenti box possono essere associati con la stessa struttura, ogni nodo può apparire in differenti contesti. Illustriamo nella figura 3.3.1-2 come viene effettuato il flattening sulla struttura di Kripke gerarchica dell'esempio precedente. La nuova struttura estesa sarà denotata con K_1^F . Notiamo che ogni nodo di K_2 appare due volte in K_1^F .

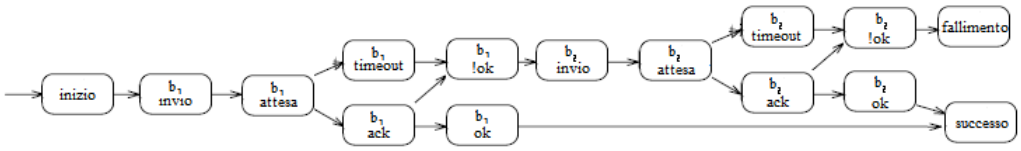

Figura 3.3.1-2

Ora diamo la definizione formale di come effettuare l'algoritmo si flattening su una struttura di Kripke gerarchica $K = < K_1, K_2, \ldots, K_n >$ [11].

Per ogni struttura $K_i = < S_i, s_0^i, R_i, L_i >$:

1. l'insieme S_i di stati di K_i^F è definito induttivamente come segue:

   - ogni nodo di K_i appartiene S_i ;
   - se u è un box di K_i con $Y_i(u) = j$ e v è uno stato di K_j^F allora (u,v) appartiene ad S_i ;

2. l'inseme R_i di transizioni di K_i^F è definito induttivamente come segue:

- per $(u,v) \in E_i$, se v è un nodo allora $(u,v) \in R_i$ altrimenti se è un box con $Y_i(v) = j$ allora $(u,(v,s_0^j)) \in R_i$;
- se w è un box di K_i con $Y_i(w) = j$ e (u,v) è una transizione di K_j^F allora ((w,u),(w,v)) appartiene ad R_i ;

3. la funzione di etichettatura $L_i : S_i \to 2^{AP}$ di K_i^F è definita induttivamente come segue:

   - se w è un nodo di K_i allora $L_i(w) = X_i(w)$;
   - se w = (u,v) dove u è un box di K_i con $Y_i(u) = j$ allora $L_i(w)$ è uguale a $L_j(v)$.

3.4 I model checker

Un model checker [18] è un tool che prende in input un modello ed una specifica, trasforma, in base al metodo di riduzione che lo caratterizza, il modello e la specifica ed infine verifica se la data specifica è vera o meno nel modello.
L'astrazione e il model checking simbolico vengono utilizzati da una buona parte dei tool di verifica in circolazione. Ogni model checker si caratterizza in base al metodo per ottimizzare il modello e in quale logica viene definita la specifica. Oltre a Yasm [12], il tool che abbiamo utilizzato e che introdurremo nel prossimo capitolo, esistono svariati tipi di model checker, eccone alcuni:

- SMV [19], è un simbolic model checker che utilizza CTL come logica temporale per definire la specifica;
- NuSMV [18], è una evoluzione di SMV, infatti questo tool dà la possibilità di definire la specifica anche in LTL;
- SLAM [20], è un model checker sviluppato da Microsoft che utilizza il paradigma CEGAR (Counter Example Guided Abstraction Refinement),

cioè quello che abbiamo analizzato nel paragrafo sul model checking astratto;

- Eureka [21], è basato sul paradigma di astrazione e raffinamento basato sui controesempi (CEGAR), ed `e una generalizzazione dell'architettura del progetto SLAM;
- BLAST [22], è un model checker realizzato a Berkeley che utilizza anch'esso l'astrazione in cui però il raffinamento è a richiesta, cioè si raffinano solo quelle regioni di codice in cui è necessaria un'astrazione più vicina al modello concreto.

Ora analizzeremo un esempio [6] su un model checker simbolico.
Utilizzeremo NuSMV per l'algoritmo di mutua esclusione di Peterson.
Nell'algoritmo di Peterson sono presenti due processi pr_0 e pr_1 che possono richiedere entrambi l'uso di una risorsa condivisa. Il sistema non deve concedere l'uso simultaneo della risorsa ma allo stesso tempo deve garantirne ad entrambi l'accesso, previa richiesta esplicita.
Un processo pr_i , nell'atto di richiedere la risorsa, segnala la richiesta ponendo una variabile booleana e_i uguale a 1 ed un'altra variabile s anch'essa booleana la pone ad i. La variabile s a differenza di e_i può essere modificata anche dal processo concorrente.
Nelle figure seguenti sono riprodotti nella sintassi NuSMV i tre moduli che eseguono l'algoritmo di Peterson.
Come si vede in figura 3.4-1 il modulo main utilizza tre variabili booleane globali:

- e_0 identifica la volontà del processo pr_0 di entrare nella sezione critica;
- e_1 identifica la volontà del processo pr_1 di entrare nella sezione critica;
- s indica il turno.

I processi pr_0 e pr_1 sono specificati mediante le variabili p_0 e p_1 che realizzano rispettivamente un'istanza del modulo pr_0 e un'istanza del modulo pr_1 . In queste due dichiarazioni vi è la parola riservata process che sta ad indicare che i processi saranno eseguiti in modo asincrono. Il modulo principale termina con l'inizializzazione delle tre variabili booleane a 0 e le dichiarazioni delle specifiche di interesse.
Le proprietà che le specifiche rappresentano sono le seguenti tre:

1. Una risorsa non può mai essere occupata sia da pr_0 che da pr_1 ;
2. Ciascun processo dopo ogni richiesta che fa deve poter ottenere la risorsa;
3. Chi si accoda per primo riceve la risorsa per primo.

```
MODULE main

VAR
s:   boolean;
e0:  boolean;
e1:  boolean;
p0:  process pr0(s,e0,e1);
p1:  process pr1(s,e0,e1);

ASSIGN
init(s)  := 0;
init(e0) := 0;
init(e1) := 0;

SPEC AG !(p0.critical & p1.critical);
SPEC AG (e0 -> AF p0.critical);
SPEC AG (e1 -> AF p1.critical);
SPEC AG (e0 & !e1 -> A [(!p1.critical) U (p0.critical)])
SPEC AG (e1 & !e0 -> A [(!p0.critical) U (p1.critical)])
```

Figura 3.4-1

Ciascuno degli altri due moduli (figure 3.4-2 e 3.4-3) dipende da tre parametri formali e manipola anche una variabile locale critical che indica lo stato di accesso della risorsa.

L'accesso alla risorsa è consentito a un processo pr_i solo in due casi:

1. e_{1-i} ha valore 0, cioè non vi è la stessa richiesta da parte del processo rivale;
2. s ha valore 1-i, cioè potrebbe esserci richiesta conflittuale ma pr_i è stato il primo ad accordarsi.

Per referenziare la variabile critical all'esterno del modulo si deve usare la notazione con 'dot', cioè $p_0.critical$ indica la variabile critical in p_0 .
Ogni istanza di un modulo secondario ha anche associata una variabile locale implicita running che rappresenta l'attivazione del processo stesso.
La variabile critical viene inizializzata in entrambi i moduli a 0.
A questo punto analizziamo nel dettaglio il modulo per pr_0 che sarà, come è facile vedere, molto simile al comportamento del modulo pr_1 .
La variabile critical viene posta, nell'istante successivo, ad 1 se e_0 vale 1 e se o e_1 vale 0 oppure s è diverso da 0. Nel caso in cui critical valeva 1 nell'istante precendente allora viene impostata a 0.
Le variabili next(s) e $next(e_0)$ sono quelle attese, c'è da osservare solamente che nell'ultimo caso della definizione di e_0 il valore non è completamente specificato perchè non è determinato in partenza se pr_0 voglia o meno entrare in sezione critica. L'intero file comprendente i tre moduli può essere eseguito in NuSMV sia in modalità batch (invia subito il risultato) oppure in modalità interattiva in cui permette all'utente di costruirsi una traccia di evoluzioni del sistema.

```
MODULE pr0(s,e0,e1)

VAR
critical :  boolean;

ASSIGN

init(critical) := 0;

next(critical) :=
case
!critical & e0 & !e1          :1;
!critical & e0 & !s=0         :1;
1                             :0;
esac;

next(e0) :=
case
critical            :0;
!critical & e0      :1;
1                   :{0,1};
esac;

next(s) :=
case
!e0 & next(e0)   :0;
1                :s;
esac;

FAIRNESS
running
```

Figura 3.4-2

```
MODULE pr1(s,e0,e1)

VAR
critical :  boolean;

ASSIGN

init(critical) := 0;

next(critical) :=
case
!critical & e1 & !e0          :1;
!critical & e1 & !s=1         :1;
1                             :0;
esac;

next(e1) :=
case
critical            :0;
!critical & e1      :1;
1                   :{0,1};
esac;

next(s) :=
case
!e1 & next(e1)   :1;
1                :s;
esac;

FAIRNESS
running
```

Figura 3.4-3

4 Yasm: il tool analizzato e le modifiche sviluppate

Yasm [13] è un model checker simbolico che si basa sul framework CEGAR. Questo tipo di approccio si può dividere in tre fasi:

1. Astrazione, qui dal modello concreto viene creata un'approssimazione;
2. Model-checking, fase di verifica vera e propria;
3. Raffinamento, viene utilizzato nel caso il risultato della verifica sia indefinito per creare un astrazione migliore.

Inizieremo questo capitolo illustrando un esempio di esecuzione sul tool Yasm per poi proseguire con la descrizione generale del codice sorgente. A quel punto non resterà che illustrare le modifiche effettuate al software spiegandone l'utilità.

4.1 Prima esecuzione sul tool Yasm

Prima di entrare nel dettaglio sulla struttura del tool vediamo come questo si comporta in fase di esecuzione con un semplice esempio.
Prendiamo il programma C di Figura 4.1-1 composto dalle funzioni main, Procedura e Procedura1.
La funzione main è rappresentata da due dichiarazioni e due successivi assegnamenti sulla variabile y da un costrutto if-else, che in entrambi i casi invocherà la funzione Procedura, ed infine dall'invocazione della funzione Procedura1 con input 8. La funzione Procedura è semplicemente composta da un if-else sullo stato della variabile locale y che nel caso del if porta all'invocazione della funzione Procedura1 e nel caso del else ad un semplice return. Infine la funzione Procedura1 è un if-else sulla variabile locale z che in entrambi i casi ritorna un valore. Il programma C rappresenta il programma da verificare ora dobbiamo definire una specifica.

```
1  void main () {
2          int y;
3          int x;
4          y = 10;
5          y = y + 1;
6          if (y == 11)
7                  y = Procedura(y);
8          else{
9                  y = y + 1;
10                 y= Procedura(8);
11         }
12         y= Procedura1(8);
13 }
14 int Procedura(int y){
15         if (y==10){
16                 y = Procedura1(y);
17                 return y;
18         }
19         else
20                 return 11;
21 }
22 int Procedura1(int z){
23         if(z>5)
24                 return 35;
25         else
26                 return 53;
27 }
```

Figura 4.1-1

La specifica in logica temporale CTL che definiamo è “EF pc = END” , cioè chiediamo se per qualsiasi percorso prima o poi il programma termina.
Preso un programma C ed una specifica in input l'esecuzione avviene posizionandosi nella directory bin e da terminale digitando il seguente comando :

```
./yasm -p 'EF pc=END' play/models/cprog/test11.c
```

Con questa riga stiamo invocando uno script di nome yasm che manderà in esecuzione il tool. Il path dopo la specifica rappresenta il percorso per arrivare al programma C.

L'output del programma è illustrato nelle figure 4.1-2, 4.1-3 e 4.1-4.

```
vadim@vadim-desktop:~/Scrivania/yasm/bin$ ./yasm -p 'EF pc=END' play/models/cprog/test11.c
ms: 64m, mx: 64m
max preds: 1
Compiling C program: play/models/cprog/test11.c

Refined Boolean Program

0/init: skip (no-line-number)
  next: 1/END
  dest: 15/main_ENTRY

1/END: goto END (no source line number)
  dest: 1/END

2/Procedura_RETURN_SELECTOR_HEAD: Return [Procedura_RETURN_0, Procedura_RETURN_1]
  next: 3/Procedura_ENTRY
  dest: 3/Procedura_ENTRY

3/Procedura_ENTRY: skip (no-line-number)
  next: 4/l5
  dest: 4/l5

4/l5: if (cond: maybe, origCond: (Procedura::y = 10)) goto Procedura1_CALL_1 else goto l9

5/Procedura1_CALL_1: prologue
  next: 6/Procedura1_RETURN_1
  dest: 10/Procedura1_ENTRY

6/Procedura1_RETURN_1: epilogue
  next: 7/l8
  dest: 7/l8

7/l8: (Procedura) return Procedura::y
  dest: 2/Procedura_RETURN_SELECTOR_HEAD

8/l9: (Procedura) return 11
  dest: 2/Procedura_RETURN_SELECTOR_HEAD

9/Procedura1_RETURN_SELECTOR_HEAD: Return [Procedura1_RETURN_0, Procedura1_RETURN_1]
  next: 10/Procedura1_ENTRY
  dest: 10/Procedura1_ENTRY

10/Procedura1_ENTRY: skip (no-line-number)
  next: 11/l12
  dest: 11/l12
```

Figura 4.1-2

```
11/l12: if (cond: maybe, origCond: (Procedura1::z > 5)) goto l13 else goto l14

12/l13: (Procedura1) return 35
  dest: 9/Procedura1_RETURN_SELECTOR_HEAD

13/l14: (Procedura1) return 53
  dest: 9/Procedura1_RETURN_SELECTOR_HEAD

14/main_RETURN_SELECTOR_HEAD: goto END (no source line number)
  next: 15/main_ENTRY
  dest: 1/END

15/main_ENTRY: skip (no-line-number)
  next: 16/l17
  dest: 16/l17

16/l17: assignment
  next: 17/l18
  dest: 17/l18

17/l18: assignment
  next: 18/l19
  dest: 18/l19

18/l19: if (cond: maybe, origCond: (main::y = 11)) goto Procedura_CALL_0 else goto l22
  next: 24/Procedura1_CALL_0
  dest: 24/Procedura1_CALL_0

19/Procedura_CALL_0: prologue
  next: 20/Procedura_RETURN_0
  dest: 3/Procedura_ENTRY

20/Procedura_RETURN_0: epilogue
  dest: 24/Procedura1_CALL_0

21/l22: assignment
  next: 22/Procedura_CALL_1
  dest: 22/Procedura_CALL_1

22/Procedura_CALL_1: prologue
  next: 23/Procedura_RETURN_1
  dest: 3/Procedura_ENTRY

23/Procedura_RETURN_1: epilogue
  dest: 24/Procedura1_CALL_0

24/Procedura1_CALL_0: prologue
  next: 25/Procedura1_RETURN_0
  dest: 10/Procedura1_ENTRY

25/Procedura1_RETURN_0: epilogue
  dest: 14/main_RETURN_SELECTOR_HEAD


Refined Boolean Program END
```

Figura 4.1-3

```
***** Model-Checking *****
Check EU: iteration: 1
Check EU: iteration: 2
Check EU: iteration: 3
Check EU: iteration: 4
Check EU: iteration: 5
Check EU: iteration: 6
Check EU: iteration: 7
Check EU: iteration: 8
Check EU: iteration: 9
Check EU: iteration: 10
Check EU: iteration: 11
Check EU: iteration: 12
Check EU: iteration: 13
Check EU: iteration: 14
Check EU: iteration: 15
Check EU: iteration: 16
Check EU: iteration: 17
Check EU: iteration: 18
Check EU: iteration: 19
Check EU: iteration: 20
Check EU: iteration: 21
Check EU: iteration: 22
Check EU: iteration: 23
Check EU: iteration: 24
Check EU: iteration: 25
Check EU: iteration: 26
Done in: 0.025s

******* SEE RESULT BELOW ***********
The property is: true
********** LOOK UP  ^ **********
Predicates used:
[]
we are done
Done in 0.531s
***********************************
STATISTICS
Overall time: 0.437s
Parsing time: 0.16s
Refine time: 0.009s
Model compilation: 0.057s
Model-Checking time: 0.025s
New predicates time: null
PredicateAbstraction time: null

QUERY: 0.001s
Predicates used:
[]
```

Figura 4.1-4

Dall'output ricaviamo le seguenti informazioni:

- il programma C viene compilato;
- il programma C viene tradotto in un programma Booleano;
- il programma Booleano viene raffinato;
- vengono effettuate svariate iterazioni per verificare la specifica;
- viene visualizzato il risultato;
- vengono analizzati il numero di predicati utilizzati e varie statistiche in funzione del tempo.

Questo è quello che un utente "vede" dall'esterno, noi nel prossimo paragrafo analizzeremo dall'interno alcune caratteristiche di Yasm.

4.2 Analisi del tool Yasm

Iniziamo la nostra analisi dalla classe YasmApp, la classe invocata dallo script yasm, che ha al suo interno il main principale del tool (figura 4.2-1). Nel main viene creato un oggetto di YasmApp e poi vengono esaminati i dati passati in input. A questo punto viene invocato il metodo run ed infine vengono stampati a video dati riguardati le statistiche ed i predicati.
Il metodo run è colui che gestisce l'esecuzione, cioè invoca i vari metodi di cui si ha bisogno per creare il modello, per settare la specifica, per costruire il model-checker e per visualizzare i risultati a video. Nel nostro percorso noi ci soffermeremo sugli aspetti riguardati la costruzione del modello.
Il frammento di codice del metodo run interessante per il nostro percorso è illustrato nella figura 4.2-2. Il primo if presente nel costrutto do-while viene utilizzato per verificare se il file c è stato settato in maniera corretta, a quel punto, nel caso la risposta sia true, viene stampato a video il path del file c. A questo punto si entra nell'if successivo solamente se sono già stati effettuati dei passi di raffinamento, in caso affermativo vengono stampati i nuovi ed i vecchi predicati.

```
  public static void main (String[] args)
{
  try
  {

    // -- create our application
    YasmApp yasmApp = new YasmApp ();

    // -- parse command line option
    yasmApp.parseCmdLine (args);


    System.err.println ("max preds: " + MAX_PREDICATES_PER_REFINEMENT);

    AlgebraValue value = yasmApp.run ();

    YasmApp.out.println ("Done in " + yasmApp.overall);

    YasmApp.out.println (yasmApp.stats);
    YasmApp.out.println ("QUERY: " + yasmApp.refiner.getVC ().sw);

    YasmApp.out.println ("Predicates used: ");
    YasmApp.out.println (yasmApp.refiner.getPredicates ());
```

Figura 4.2-1

```
public AlgebraValue run () throws CTLNodeParserException,
                                  PProgram.ParseException,
                                  FileNotFoundException
{

  stats.start ();

  AlgebraValue value = null;
  MvSet result = null;

  do
    {
      if (getCFile ())
        {
          YasmApp.out.println ("Compiling C program: " + bpFile);
          if (refiner != null)
            {
              YasmApp.out.println ("Old predicates: " +
                                   refiner.getPredicates ());
              YasmApp.out.println ("New predicates: " +
                                   refiner.getNewPredicates ());
            }

          compileCProgram ();
        }
```

Figura 4.2-2

L'ultimo statement del metodo run interessante per la nostra analisi è l'invocazione del metodo compileCProgram.
Questo metodo crea, se c'è ne bisogno, un'oggetto compiler ed un oggetto pProgram nel seguente modo:

```
if (compiler == null)
  compiler = new PProgramCompiler (useHyperEdges,
                                   unknownInit);
if (pProgram == null)
  {
    stats.startParse ();
    pProgram = PProgram.parse (new NullExprAbstractor (fac), bpFile,
                               selectorType, stmtBlocking);

    stats.stopParse ();
  }
```

Figura 4.2-3

La creazione dell'oggetto pProgram, vista la sua importanza, verrà analizzata nel dettaglio dopo aver terminato l'analisi del metodo attualmente in esame.
Dopo aver valutato l'esistenza di un refiner viene effettuato il raffinamento sul pProgram e successivamente vi è un processo di stampa a video del pProgram raffinato tramite il metodo print. L'ultima cosa di rilevante importanza è l'invocazione del metodo setXKripkeStructure che ha come unico parametro l'invocazione del metodo compile della classe PProgramCompiler. Le ultime istruzioni appena descritte sono illustrate nella figura 4.2-4.

```
stats.startRefine ();
pProgram = refiner.doProgramRefine (pProgram);
stats.stopRefine ();

System.err.println ();
System.err.println ("Refined Boolean Program");
System.err.println ();
pProgram.print (YasmApp.err);
System.err.println ();
System.err.println ("Refined Boolean Program END");
System.err.println ();

PProgramCompiler pCompiler = (PProgramCompiler)compiler;

pCompiler.setPProgram (pProgram);
// XXX just for now
pCompiler.setMaxDDVars (100);
stats.startCompile ();
setXKripkeStructure (pCompiler.compile ());
stats.stopCompile ();
```

Figura 4.2-4

4.2.1 Descrizione dettagliata del PProgram

La classe PProgram esprime il predicate program [14] o programma Booleano e rappresenta la prima trasformazione del nostro programma C passato in input al tool Yasm. Prendiamo in esame il seguente programma C [14] sintetico:

```
1   int main (void) {
2     /* ... */
3     fun();
4     /* ... */
5     return 0;
6   }
7
8   int fun (void) {
9     /* ... */
10    return 0;
11  }
```

Figura 4.2.1-1

Le righe con /*...*/ possono rappresentare un qualsiasi assegnamento o dichiarazione. Il PProgram corrispondente è mostrato in figura 4.2.1-2 [14].
Ogni nodo del PProgram è rappresentato da un oggetto della classe PStmt, i nodi init ed end rappresentano l'inizio e la fine del programma ed hanno le stesse caratteristiche con qualsiasi programma C in input. Ogni Pstmt [14] ha i seguenti attributi rilevanti :

- una stringa che specifica l'etichetta dello statement;
- un riferimento ad un altro statement che rappresenta il successore sintattico chiamato next (rappresentato in figura con una freccia continua);
- un riferimento ad un altro statement che rappresenta il successore nel control-flow chiamato dest (rappresentato in figura con una freccia tratteggiata).

Per rendere ancora più chiare le idee valutiamo il PProgram delle figure 4.1-2 e 4.1-3. In questo caso le informazioni sono descritte in forma sequenziale , ma è molto semplice da questi dati ricavare una rappresentazione grafica. Vi sono 26 statement ed ognuno di essi è rappresentato dalla sua etichetta, da un puntatore a next ed un puntatore a dest (figura 4.2.1-3).

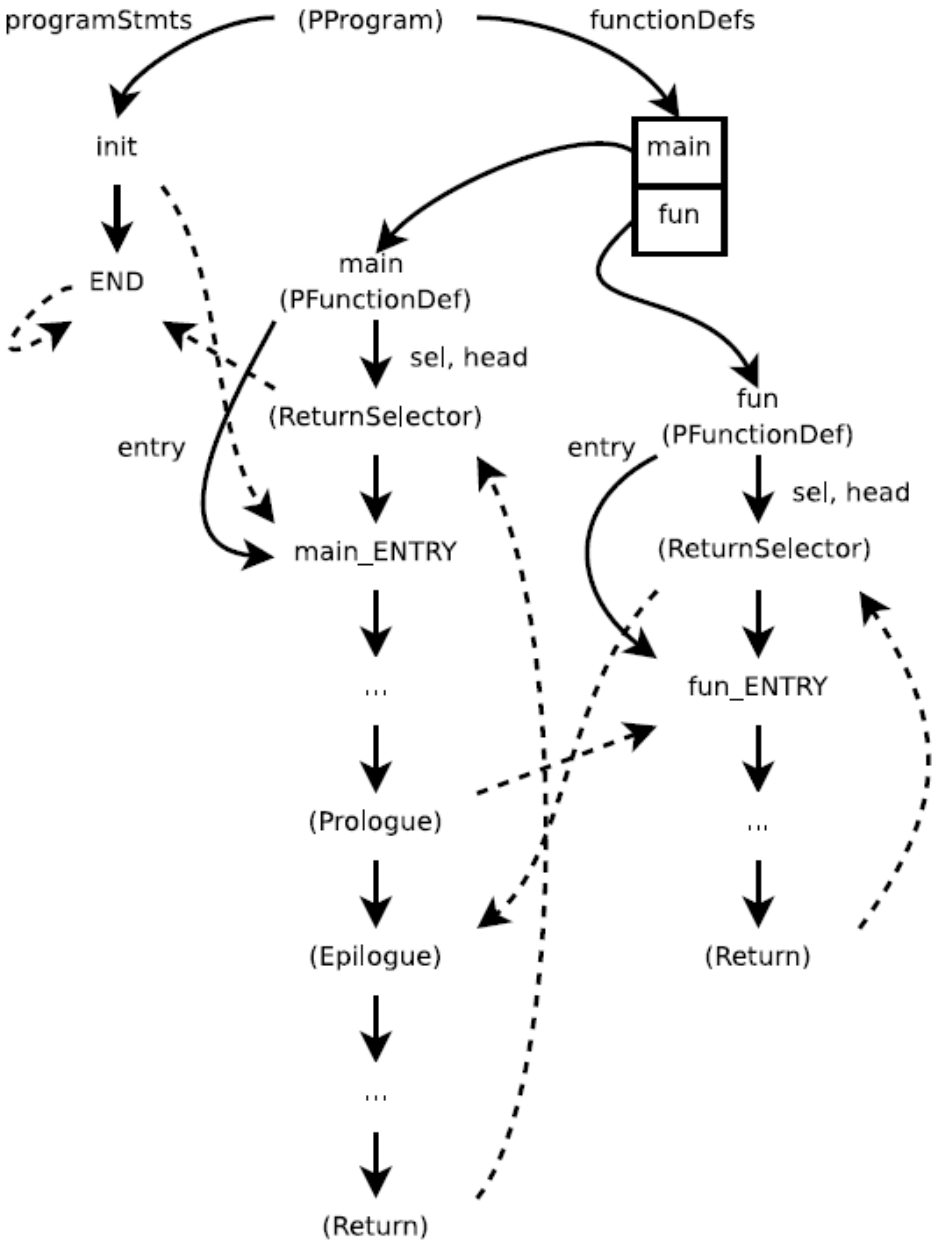
programStmts
(PProgram)
functionDefs
init
main
fun
END
main
(PFunctionDef)
sel, head
(ReturnSelector)
fun
(PFunctionDef)
entry
entry
sel, head
main_ENTRY
(ReturnSelector)
...
fun_ENTRY
(Prologue)
...
(Epilogue)
(Return)
...
(Return)

Figura 4.2.1-2

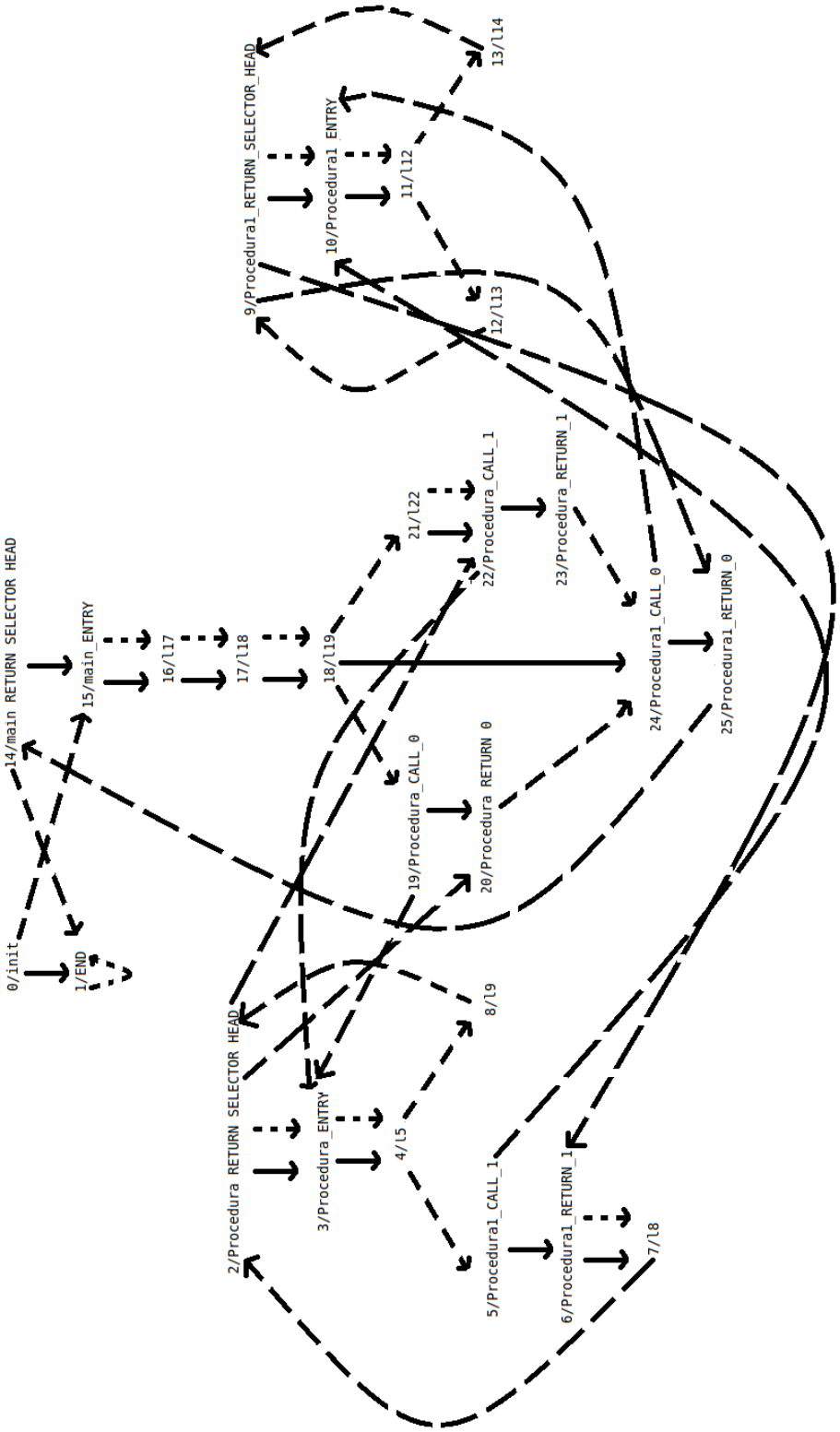
0/init
1/END
14/main RETURN SELECTOR HEAD
15/main_ENTRY
16/l17
17/l18
18/l19
21/l22
22/Procedura_CALL_1
23/Procedura_RETURN_1
24/Procedura1_CALL_0
25/Procedura1_RETURN_0
19/Procedura_CALL_0
20/Procedura RETURN 0
8/l9
2/Procedura RETURN SELECTOR HEAD
3/Procedura_ENTRY
4/l5
5/Procedura1_CALL_1
6/Procedura1_RETURN_1
7/l8
9/Procedura1_RETURN_SELECTOR_HEAD
10/Procedura1 ENTRY
11/l12
12/l13
13/l14

Figura 4.2.1-3

4.2.2 Ulteriori considerazioni sull'analisi

Riprendiamo il nostro percorso all'interno di Yasm trattando ora il metodo compile. Il metodo in esame è così definito:

```
public XKripkeStructure compile ()
{
  seal ();
  CFA cfa = buildCFA ();

  cfaMvSetFactory = new CFAMvSetFactory (mvSetFactory, cfa);
  trans = cfaMvSetFactory.getMvRelation ();

  // XXX figure out what init is, for now any variable assignment
  // XXX is deemed possible
  MvSet initStates = mvSetFactory.top ();

  // -- if there is an invariant we must intersect init with it
  if (invar != null)
      initStates = initStates.and (invar);

  init = cfaMvSetFactory.embed (0, initStates);
  PredicateTable pTable = pProgram.getPredTable ();

  return new XKripkeStructure (trans,
      init,
      preToPostMap,
      cfaMvSetFactory.embedMvSet (postVarCube),
      cfaMvSetFactory.embedMvSet (preVarCube),
      null,
      algebra,
      pTable.getNumVars (),
      pTable.getNumVars (),
      getCtlReWriter (),
      getStatePresenter ());
}
```

Figura 4.2.2-1

Questo metodo tramite la creazione di un oggetto della classe CFA, che tratteremo in dettaglio nel prossimo paragrafo, ed un oggetto della classe CFAMvSetFactory riuscirà a creare un oggetto della classe XKripkeStructure.

Un oggetto della classe XKripkeStructure è la struttura su cui si effettua la verifica del modello.

4.2.3 Descrizione dettagliata del CFA

Nel precedente paragrafo abbiamo visto che veniva creato un oggetto della classe CFA, ma non abbiamo valutato come è stato creato. Ora analizziamo prima le caratteristiche della classe CFA e poi valutiamo il metodo buildCFA.
Un oggetto della classe CFA è un grafo, e come tale è rappresentato da un insieme di nodi e un insieme di archi che connettono tra loro i nodi.
Gli attributi che lo rappresentano sono i seguenti:

```
public class CFA
{
  CFANode[] nodes;
  int used = 0;
  List fwdEdges;
  List bwdEdges;
```

Figura 4.2.3-1

L'array di tipo CFANode rappresenta i nodi, l'intero used rappresenta il numero di nodi ed infine le due liste rappresentano gli archi entranti (bwd) e gli archi uscenti (fwd). Un oggetto della classe CFANode identifica un singolo nodo, ogni nodo ha i seguenti attributi:

```
public class CFANode
{
  int id;
  String strValue;
  MvSet state;
```

Figura 4.2.3-2

L'id è l'identificativo del nodo, la stringa è l'etichettatura del nodo e state rappresenta il valore di verità del nodo.
Ogni elemento delle liste fwdEdges e bwdEdges è una istanza della classe CFAEdge che ha un campo stringa, un intero sourceId che identifica la sorgente da cui parte l'arco, due interi destId e destId2 che rappresentano le destinazioni

(destId2 può anche essere vuoto) ed un MvRelation che rappresenta il valore di verità dell'arco (figura 4.2.3-3).

```
public class CFAEdge
{
  String strValue;
  int sourceId;
  int destId;
  int destId2;
  MvRelation data;
```

Figura 4.2.3-3

Queste sono le caratteristiche principali di un oggetto della classe CFA, ora possiamo illustrare tramite il metodo buildCFA come Yasm crea effettivamente questo oggetto. Nel corpo del metodo (figura 4.2.3-4) si crea un oggetto cfa il cui parametro del costruttore rappresenta il numero di statement del pProgram.
Ora viene effettuato un for su tutti gli statement del pProgram e per ognuno di essi viene creato un nodo col metodo addNode, i dati passati a addNode sono l'etichetta dello statement e un valore di verità uguale a true.
Come ultima operazione vi è un nuovo for su tutti gli statement del pProgram che questa volta serve per creare gli archi. Qui in base al tipo di statement viene invocato uno specifico metodo.

```
private CFA buildCFA ()
{
  // XXX Should compute CFG size here (when function calls are handled)

  if (pProgram.getInconsistent () != null)
    invar = exprToMvSet (pProgram.getInconsistent ()).eq (mvT).not ();

  // -- create the CFA
  CFA cfa = new CFA (pProgram.getStmtList ().size ());

  // -- add nodes
  for (Iterator it = pProgram.getStmtList ().iterator (); it.hasNext ();)
    cfa.addNode (((PStmt) it.next ()).getLabel (), mvSetFactory.top ());

  // -- now we need to add the edges
  for (Iterator it = pProgram.getStmtList ().iterator (); it.hasNext ();)
  {
    PStmt pStmt = (PStmt) it.next ();
    if (pStmt instanceof FunctionCallPrologue)
      handleFunctionCall (cfa, (FunctionCallPrologue) pStmt);
    else if (pStmt instanceof ReturnSelectorPStmt)
      handleReturnSelector (cfa, (ReturnSelectorPStmt) pStmt);
    else if (pStmt instanceof SkipPStmt)
      handleSkip (cfa, (SkipPStmt) pStmt);
    else if (pStmt instanceof IfPStmt)
      handleIf (cfa, (IfPStmt) pStmt);
    else if (pStmt instanceof PrllAsmtPStmt)
      handleAssign (cfa, (PrllAsmtPStmt) pStmt);
    else if (pStmt instanceof GotoPStmt)
      handleGoto (cfa, (GotoPStmt) pStmt);
    else if (pStmt instanceof NDGotoPStmt)
      handleNDGoto (cfa, (NDGotoPStmt) pStmt);
    else
      throw new RuntimeException ("Unable to handle " + pStmt.getClass ());
  }
  return cfa;
}
```

Figura 4.2.3-4

4.2.4 Descrizione di una XKripkeStructure

Una XKripke Structure [23] è un modello $K = < L, AP, S, s_0, R, I >$, dove:

- $L = (\int, \cup, \cap, \neg)$ è un algebra;
- AP è l'insieme di proposizioni atomiche che hanno valori in L;
- S è un insieme finito di stati;
- $s_0 \in S$ è uno stato iniziale;

- $R : S \times S \rightarrow \int$ è una relazione di transizione multi-valore;
- $I : S \rightarrow (AP \rightarrow \int)$ è una funzione di etichettatura che mappa gli stati in s nell'insieme degli L-valori su AP;

$\int$ è una logica di Belnap, questa logica ha 4 valori di verità [23] come rappresentato nel reticolo di figura 4.2.4-1. La verità di una proposizione atomica è rappresentata da TT mentre la falsità è rappresentata da FF.

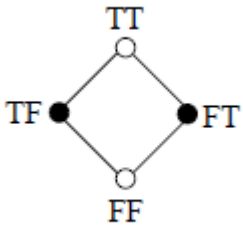

Figura 4.2.4-1

Un esempio di una Xkripke [23] può essere il seguente:

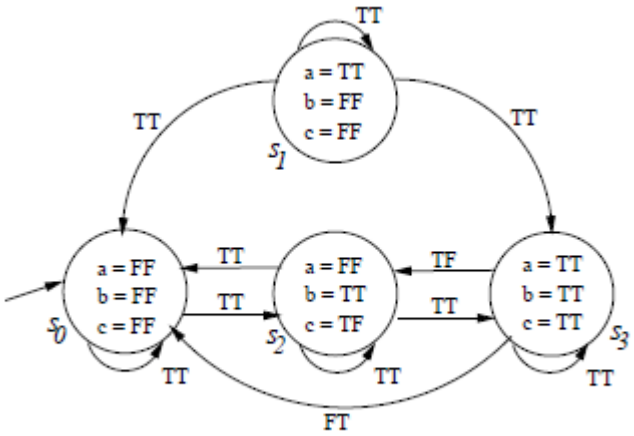

Figura 4.2.4-2

4.3 Descrizione dell'obbiettivo e modifiche effettuate

L'obbiettivo delle modifiche effettuate è stato quello di produrre un modello gerarchico. Le modifiche quindi verteranno sulla trasformazione del pProgram, del CFA e di conseguenza della XKripkeStructure. Visto che la parte riguardate la verifica non è stata trattata, eseguiremo un algoritmo di flattening per produrre nuovamente le strutture lineari ed effettuare il model checking come in precedenza (figura 4.3-1). A questo punto confronteremo brevemente gli output del tool Yasm con quello da noi modificato per valutare se il nuovo tool è coerente e quanto tempo impiega in più dell'originale.

4.3.1 pProgram gerarchico

Creare un pProgram gerarchico significa spezzare un pProgram flat in più parti. Ogni pProgram rappresenterà una funzione del programma dato in input. La cosa più importante è che questi pProgram non potranno essere collegati fra loro, dovranno rappresentare entità separate. Per ogni chiamata di funzione verrà creato un nuovo tipo di statement con le seguenti caratterisitche:

Box_”nome funzione”_”numero invocazione”

La parte esterna alle virgolette sarà uguale per tutte le chiamate a funzione, mentre quello tra virgolette rappresenterà il nome della funzione e infine un indice che identifica una specifica chiamata.

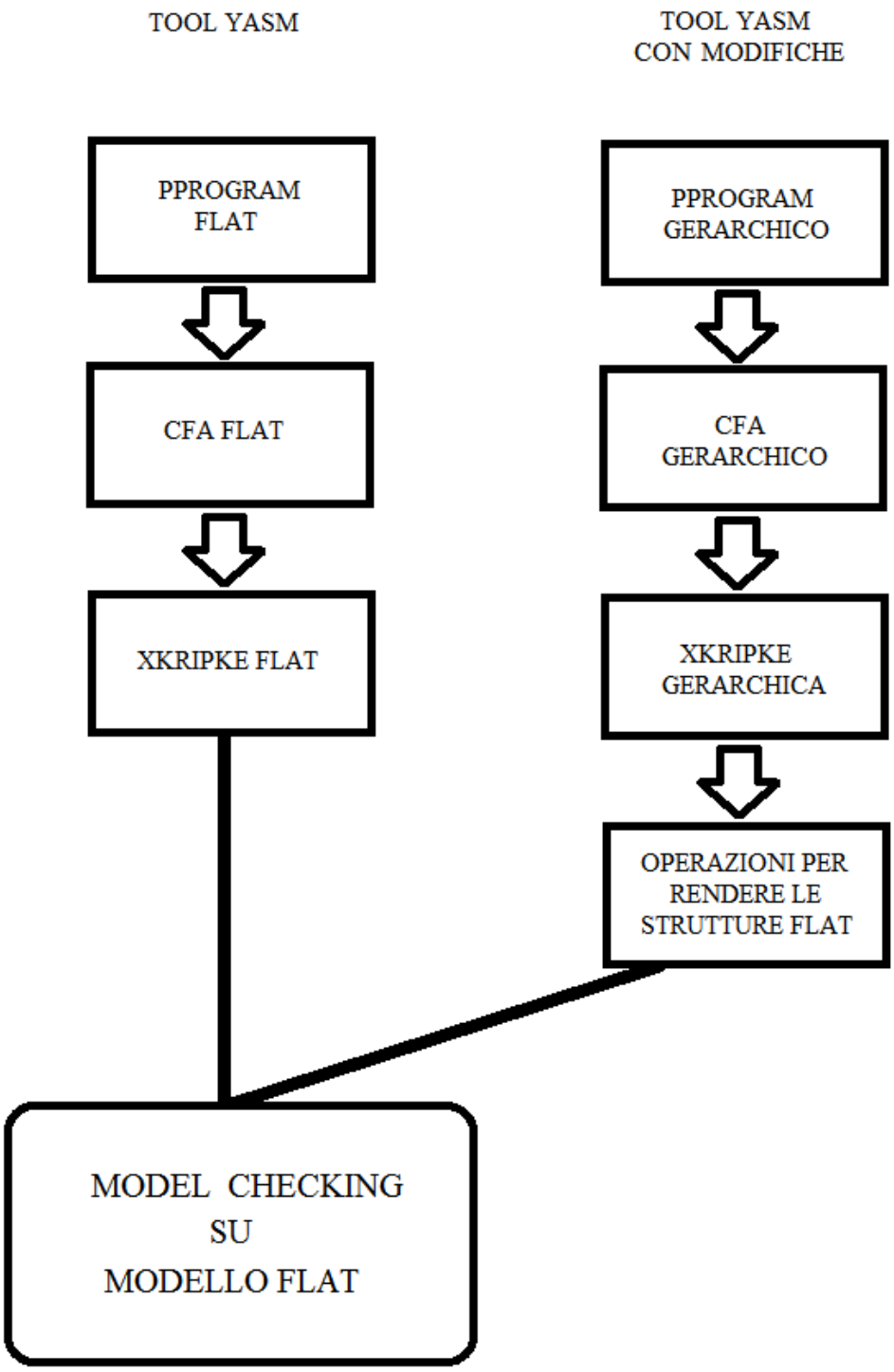
TOOL YASM
TOOL YASM
CON MODIFICHE
PPROGRAM
FLAT
PPROGRAM
GERARCHICO
CFA FLAT
CFA
GERARCHICO
XKRIPKE FLAT
XKRIPKE
GERARCHICA
OPERAZIONI PER
RENDERE LE
STRUTTURE FLAT
MODEL CHECKING
SU
MODELLO FLAT

Figura 4.3-1

Per creare questi tipi di pProgram prima di tutto abbiamo dovuto modificare la fase di compilazione del codice C. Questo perchè se prima invocavamo una procedura ed il corpo di questa non era presente nel file veniva generato un errore, ma ora è necessario avere una situazione del genere visto che dobbiamo creare più pProgram da frammenti dello stesso codice C. L'ultima modifica rilevante è stata quella di produrre il nuovo statement Box accennato in precedenza. Per fare questo ad ogni chiamata a procedura abbiamo modificato il flusso di creazione di statement sostituendo ai due statement CALL e RETURN un unico statement Box che ha gli stessi archi entranti di CALL e gli stessi archi uscenti di RETURN. I pProgram gerarchici di Procedura e main del programma C del paragrafo 4.1 sono presentati nelle figure 4.3.1-1, 4.3.1-2.

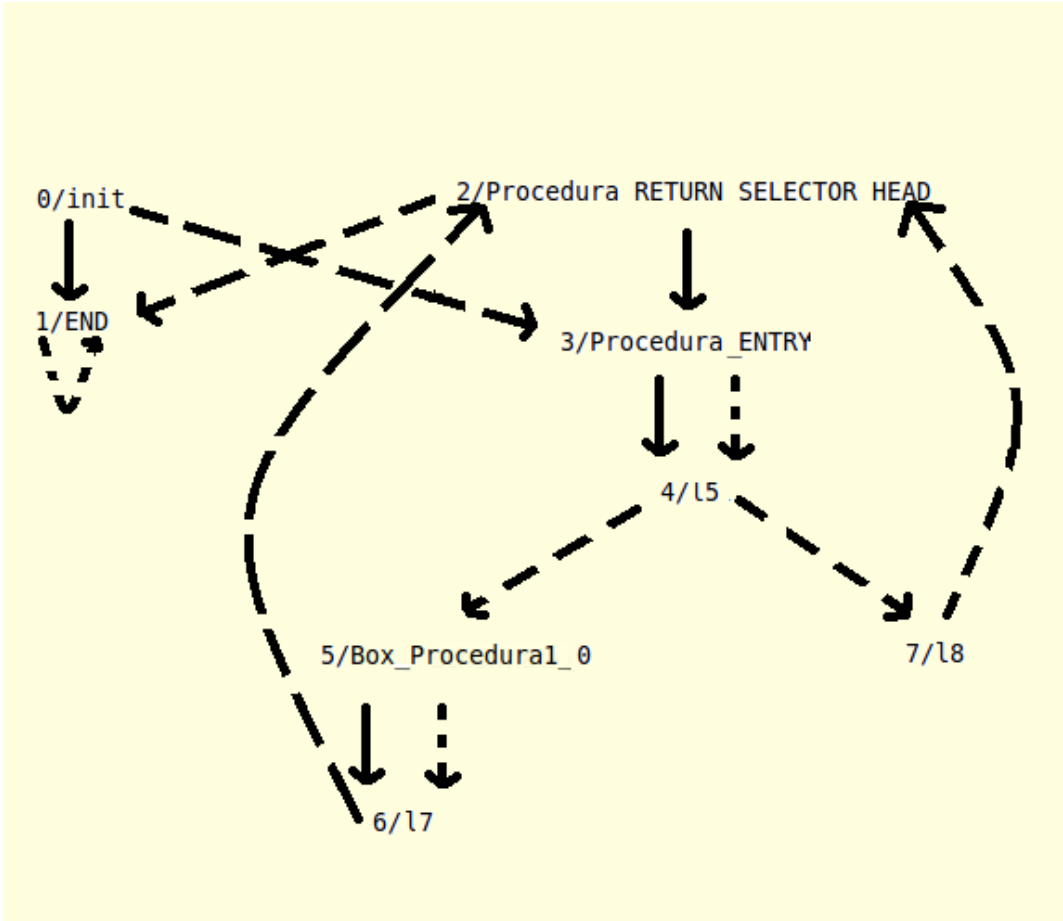

Figura 4.3.1-1

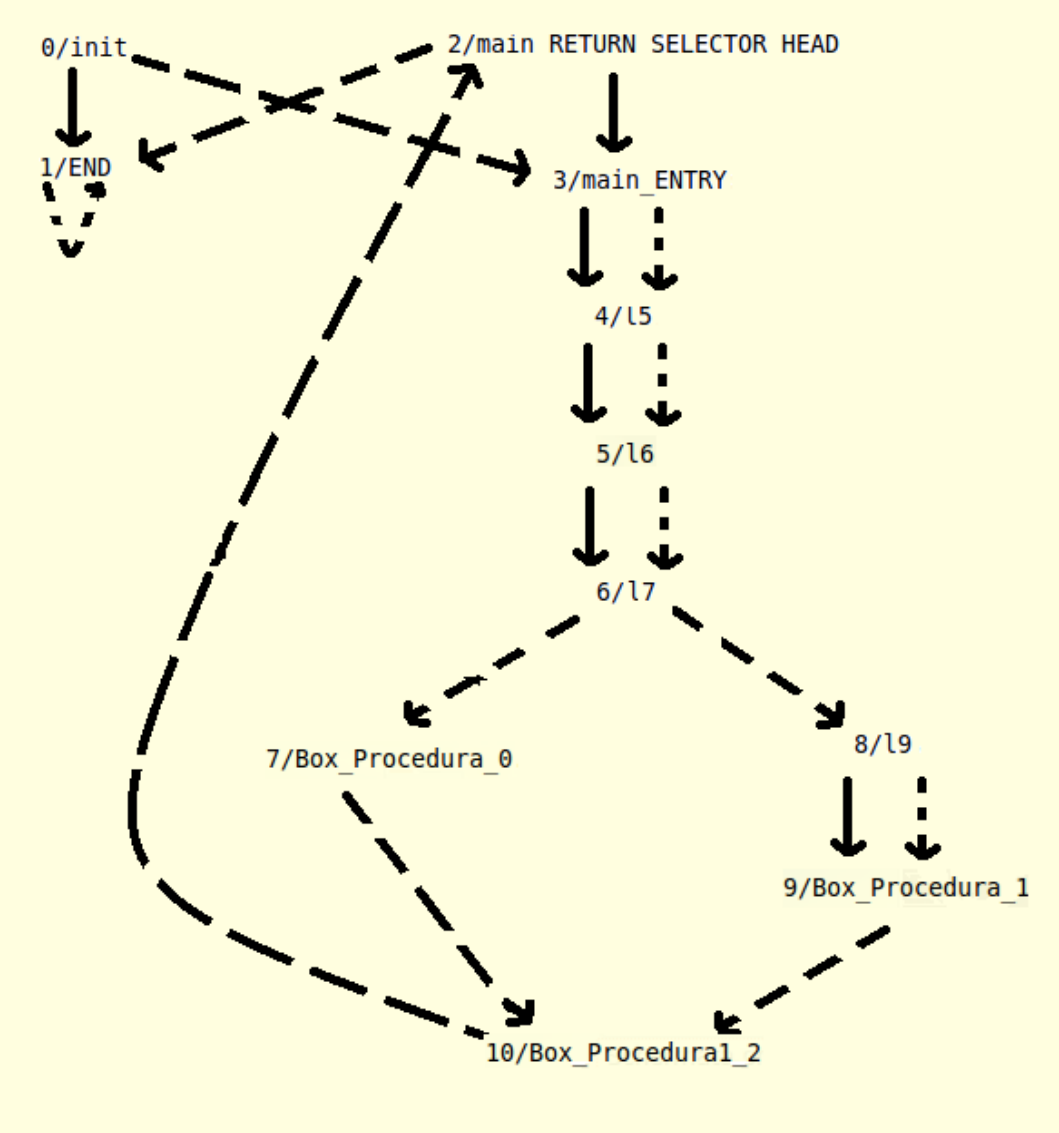

Figura 4.3.1-2

4.3.2 CFA gerarchico

Anche qui per creare un CFA gerarchico abbiamo creato vari CFA ognuno che descrivesse una specifica funzione. In questo caso visto che trattiamo con una specie di grafo e che ogni statement è rappresentato da un'istanza di CFANode, dobbiamo creare una nuova classe CFABox per quegli statement con prefisso Box. Le modifiche principali che sono state eseguite sono presenti nel metodo

buildCFA descritto nella sezione 4.2.3. Abbiamo dovuto inserire un if-else per l'aggiunta dei nodi in cui si valutava se lo statement rappresentava o meno un box, e per gli archi abbiamo dovuto modificare la parte relativa agli statement del tipo ReturnSelector perchè ora ogni nodo box dove puntare, come detto in precedenza, a quello che puntava lo statement RETURN.

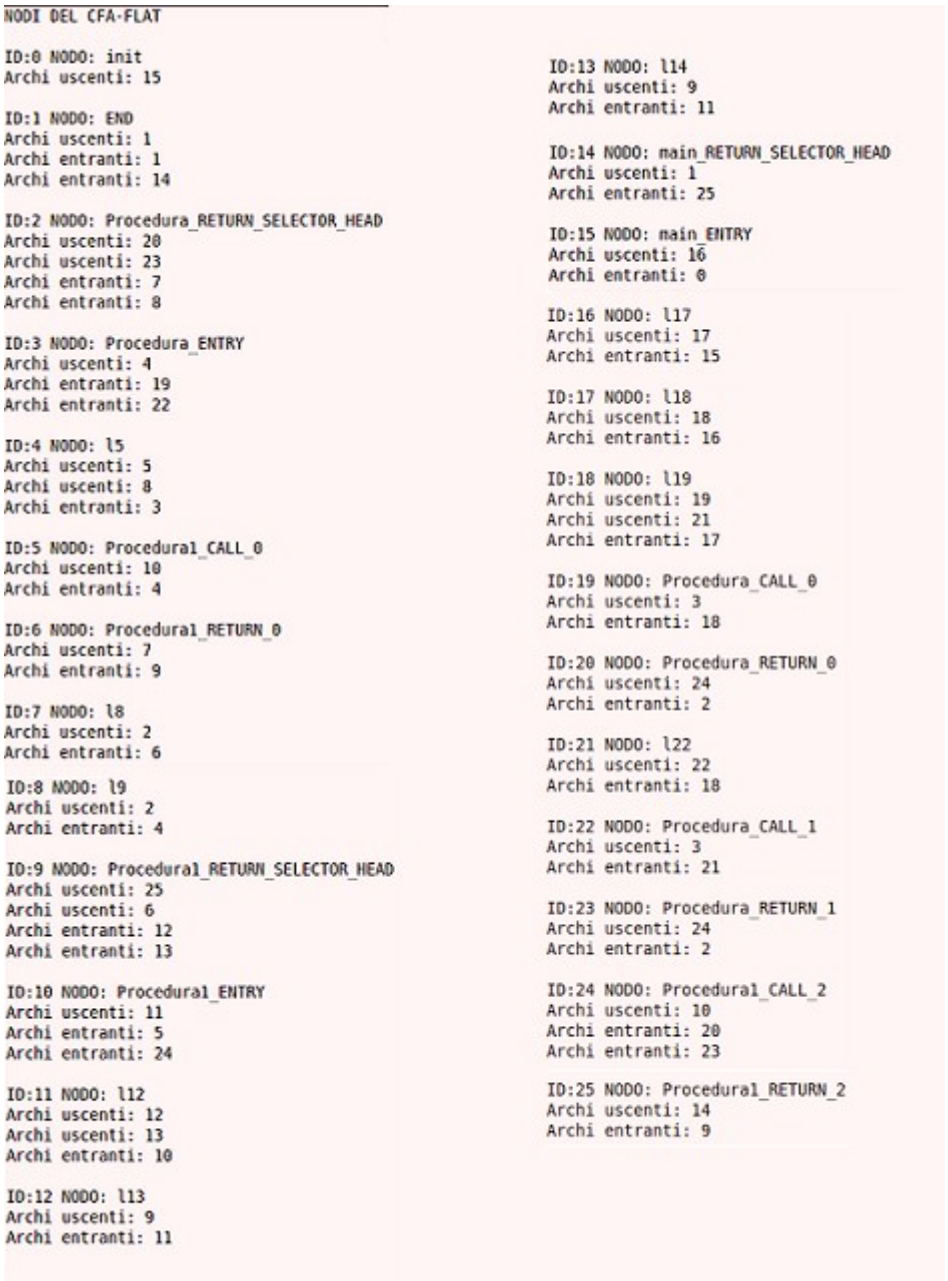

Figura 4.3.2-1

Per chiarire meglio illustriamo nella figura 4.3.2-1 il CFA del programma C del paragrafo 4.1 e nelle successive figure i CFA corrispondenti per una strutturazione gerarchica.

```
NODI E BOX DEL CFA main

Nodo: 0 Label: init
Archi uscenti: 3

Nodo: 1 Label: END
Archi uscenti: 1
Archi entranti: 1
Archi entranti: 2

Nodo: 2 Label: main_RETURN_SELECTOR_HEAD
Archi uscenti: 1
Archi entranti: 10

Nodo: 3 Label: main_ENTRY
Archi uscenti: 4
Archi entranti: 0

Nodo: 4 Label: l5
Archi uscenti: 5
Archi entranti: 3

Nodo: 5 Label: l6
Archi uscenti: 6
Archi entranti: 4

Nodo: 6 Label: l7
Archi uscenti: 7
Archi uscenti: 8
Archi entranti: 5

Box: 7 Label: Box_Procedura_0
Archi uscenti: 10
Archi entranti: 6

Nodo: 8 Label: l9
Archi uscenti: 9
Archi entranti: 6

Box: 9 Label: Box_Procedura_1
Archi uscenti: 10
Archi entranti: 8

Box: 10 Label: Box_Procedura1_2
Archi uscenti: 2
Archi entranti: 7
Archi entranti: 9
```

Figura 4.3.2-2

Notiamo che il numero di nodi complessivo è molto simile ad esempio in questo caso sia per il CFA-flat che per CFA-gerarchico sono 26. Questo però è veramente un "caso", infatti col gerarchico nel complesso si riducono gli stati con un maggior numero di invocazioni della stessa funzione viceversa aumentano se vi sono molte funzioni invocate poche volte. Questo perchè col gerarchico si recupera uno stato ad ogni chiamata di procedura, un box sostituisce i nodi call e return, mentre si aggiungono due stati, init ed end, per ogni funzione distinta.

```
NODI E BOX DEL CFA Procedura

Nodo: 0 Label: init
Archi uscenti: 3

Nodo: 1 Label: END
Archi uscenti: 1
Archi entranti: 1
Archi entranti: 2

Nodo: 2 Label: Procedura_RETURN_SELECTOR_HEAD
Archi uscenti: 1
Archi entranti: 6
Archi entranti: 7

Nodo: 3 Label: Procedura_ENTRY
Archi uscenti: 4
Archi entranti: 0

Nodo: 4 Label: l5
Archi uscenti: 5
Archi uscenti: 7
Archi entranti: 3

Box: 5 Label: Box_Procedura1_0
Archi uscenti: 6
Archi entranti: 4

Nodo: 6 Label: l7
Archi uscenti: 2
Archi entranti: 5

Nodo: 7 Label: l8
Archi uscenti: 2
Archi entranti: 4
```

```
NODI E BOX DEL CFA Procedura1

Nodo: 0 Label: init
Archi uscenti: 3

Nodo: 1 Label: END
Archi uscenti: 1
Archi entranti: 1
Archi entranti: 2

Nodo: 2 Label: Procedura1_RETURN_SELECTOR_HEAD
Archi uscenti: 1
Archi entranti: 5
Archi entranti: 6

Nodo: 3 Label: Procedura1_ENTRY
Archi uscenti: 4
Archi entranti: 0

Nodo: 4 Label: l5
Archi uscenti: 5
Archi uscenti: 6
Archi entranti: 3

Nodo: 5 Label: l6
Archi uscenti: 2
Archi entranti: 4

Nodo: 6 Label: l7
Archi uscenti: 2
Archi entranti: 4
```

Figura 4.3.2-3

4.3.3 Ultime modifiche effettuate

Dopo la creazione dei vari cfa sono state create tutte le xkripke corrispondenti allo stesso modo di come venivano create le xkripke flat. Ora ci manca solamente riorganizzare tutte le strutture in modo tale da renderle flat e passarle al model-checker.

Per quanto riguarda il pProgram abbiamo creato, nella fase di compilazione del codice, un pProgram-flat, quindi non abbiamo inserito particolari algoritmi. Per i cfa abbiamo effettuato un algoritmo di flattening sui nodi ed abbiamo acquisito tutte le informazioni degli archi dal pProgram-flat. L'algoritmo utilizzato per contare il numero di nodi necessari è rappresentato nella figura 4.3.3-1.

```
public int contaNodi(){
     int length = 0;
     for(int i=0;i<cfa.size();i++){
             CFA c = (CFA) cfa.get(i);
             ArrayList l = c.getNodes();
             length = length + (l.size()-2);
             //System.out.println("size cfa: "+c.nodeSize());
             for(int j=0;j<l.size();j++){
                     if( l.get(j) instanceof CFA.CFANode ){
                             CFA.CFANode n = (CFA.CFANode) l.get(j);
                             //System.out.println("NODO: "+n.getStrValue());
                     }
                     else{
                             CFA.CFABox b = (CFA.CFABox) l.get(j);
                             //System.out.println("Box: "+b.getStrValue());
                             length = length+1;
                     }
             }
     }
     return length+2;
}
```

Figura 4.3.3-1

Il metodo contaNodi è importante perchè come dicevamo in precedenza non vi è una corrispondenza 1 a 1 tra gli statemente di un cfa-flat e un insieme di cfa-gerarchici. Viene effettuato un for sul numero di cfa presenti in una lista, da qui viene inserito alla variabile length la lunghezza di ogni cfa scansionato -2 (per gli stati init ed end). Il secondo for serve per valutare se l'elemento del cfa-gerarchico

in esame è un nodo oppure un box, perchè nel caso sia un box bisogna aggiungere uno alla variabile length. Alla fine ritorniamo length+2 per valutare gli stati init ed end che devono essere collegati al main.
L'algoritmo di flattening per i nodi è rappresentato nella figura 4.3.3-2.
Questo algoritmo è stato effettuato in maniera ricorsiva e quindi come tale viene effettuato un controllo sul fatto che i nodi del cfa-gerarchico siano maggiori dell'indice in analisi. I dati in input rappresentano il cfa-gerarchico in analisi e l'indice sotto esame .A questo punto ci siano trovati in una situazione per casi basata sulle caratteristiche dei vertici. Il primo if spezza l'analisi su vertici di tipo nodo e box. Se ci troviamo in un vertice di tipo nodo dobbiamo solamente distinguere quale nodo stiamo analizzando, è invece interessante descrivere l'if-else presente nelle parte riferita ai box. Qui notiamo un metodo trova con input una stringa, la stringa rappresenta il nome del cfa, questo serve per vedere se è stato valutato o meno il cfa nella creazione dei nodi.
Creato il cfa-flat ci è bastato creare una nuova xkripke che acquisisse i dati necessari dal pProgram-flat e dal cfa-flat e darlo in pasto al model-checker.
Il tool modificato risponde ad ogni problema posto allo stesso modo del tool Yasm sia che ci sia del raffinamento con aggiunta di predicati che in caso contrario. L'unica cosa che cambia sono i tempi di risposta che comunque sono accettabili, infatti per un buon numero di prove, tutte rappresentanti situazioni diverse e di interesse, il tempo peggiore di risposta si aggira sui 4-5 decimi si secondo al di sopra del tool Yasm.

```
public void setCFA(CFA c , int index ){
     boolean entry = false;
     if(c.nodeSize() > index ){
             if( c.getNode(index) instanceof CFA.CFANode ){
                     CFA.CFANode n = (CFA.CFANode) c.getNode(index);
                     if( n.getStrValue().equals("main_RETURN_SELECTOR_HEAD") ){
                             if(cfa.size() > 1){
                                     CFA cc = (CFA) cfa.get(1);
                                     CFA.CFANode nn = (CFA.CFANode) cc.getNode(2);
                                     String ss = nn.getStrValue();
                                     String[] sst = ss.split("_");
                                     ss = sst[0];
                                     stringhe.add(ss);
                                     setCFA( cc , 2 );
                             }
                     }
                     if( n.getStrValue().startsWith("l") ){
                             flatCFA.addNode( "l"+contatore, mvSetFactory.top() );
                             contatore++;

                     }
                     else{
                             flatCFA.addNode( n.getStrValue(), mvSetFactory.top() );
                             contatore++;

                     }
                     setCFA( c , index+1 );
             }
             else{
                     CFA.CFABox b = (CFA.CFABox) c.getNode(index) ;
                     String s = b.getStrValue();
                     mappa.put( s, (ArrayList) c.getFwdEdges(b) );
                     String st;
                     String [] str;
                     str = s.split("_");
                     s = str[1];
                     if( !trova(s) ){
                             st = s+"_CALL_"+str[2];
                             flatCFA.addNode(s+"_CALL_"+str[2], mvSetFactory.top() );
                             contatore++;
                             st = s+"_RETURN_"+str[2];
                             flatCFA.addNode(s+"_RETURN_"+str[2], mvSetFactory.top() );
                             contatore++;

                             entry = true;
                     }
                     else{

                             st = s+"_CALL_"+str[2];
                             flatCFA.addNode(s+"_CALL_"+str[2], mvSetFactory.top() );
                             contatore++;
                             st = s+"_RETURN_"+str[2];
                             flatCFA.addNode(s+"_RETURN_"+str[2], mvSetFactory.top() );
                             contatore++;

                     }
                     stringhe.add(s);
                     st_ret.add(st);
                     setCFA( c , index+1 );
                     if( entry )
                             setCFA( cercaCFA(s) , 2 );

             }
     }
}
```

Figura 4.3.3-2

5 Conclusioni

L'obbiettivo della nostra tesi è stato quello di estendere YASM in modo da permettere la verifica di sistemi software gerarchici utilizzando l'approccio dell'appiattimento del modello. In pratica, dal software gerarchico, si produce dapprima un modello gerarchico che poi viene appiattito in un modello classico utilizzando un algoritmo di appiattimento. Infine, sul modello classico prodotto viene poi fatto girare il tool Yasm.
Per raggiungere questo obbiettivo abbiamo dovuto percorrere studiare e/o implementare diversi strumenti, tra cui:

- modellare un sistema critico gerarchico;
- modellare un sistema tramite una struttura di Kripke;
- definire formalmente una specifica;
- analizzare le logiche temporali LTL e CTL;
- effettuare la verifica formale di un modello rispetto ad una data specifica;
- risolvere il problema di model-checking tramite automi;
- analizzare il problema dell'esplosione degli stati e valutare vari metodi per ridurre questo problema;
- comprensione dettagliata di come il tool Yasm produce un modello.

Fatto tutto questo siamo riusciti a raggiungere l'obbiettivo della nostra tesi che rappresenta un primo passo verso l'implementazione di un model-checker gerarchico completo ed efficiente. In pratica, la soluzione del flattening, sebbene funzionante è solitamente la meno efficiente. Infatti è stato mostrato da un punto di vista teorico che un algoritmo diretto sul modello gerarchico dovrebbe essere più efficiente. Infatti, nel paragrafo 3.3 riguardante il model checking gerarchico abbiamo potuto notare che un modello gerarchico è esponenzialmente più succinto rispetto al corrispondente modello flat. In [11] è stato dimostrato che la

complessità del model checking su un modello gerarchico con specifica in LTL è lineare rispetto al modello ed esponenziale rispetto alla formula. In [24] è stato inoltre dimostrato che la complessità del model checking su un modello gerarchico con specifica in CTL è sempre esponenziale sulla formula mentre sul modello è esponenziale sono rispetto alcuni parametri del modello (il numero degli exit-node) che solitamente sono piccoli rispetto al resto del modello.

Sapendo che con un modello flat la complessità risulterebbe esponenziale sul modello ci rendiamo conto che creare un model-checker gerarchico risulta essere un ottima scelta per migliorare le prestazioni.

Illustriamo in figura 5-1, facendo anche riferimento alla figura 4.3-1, quale dovrebbe essere la situazione finale di un tool efficiente di model checking gerarchico, con l'aggiunta di un model checker che lavori direttamente sul modello gerarchico.

Si noti che, il model checker prodotto in questa tesi oltre ad essere utile di per sé (visto che permette di verificare la correttezza di sistemi gerarchici implementando un algoritmo noto in letteratura), sarà utile per valutare (confrontare) sperimentalmente il vantaggio che se ne deriva dall'utilizzare un model checker gerarchico puro.

Ritornando alla figura 5-1, i blocchi e le transizioni di colore blu sono quelle da noi effettuati e che verranno utilizzati anche nel model checker gerarchico da implementare.

I blocchi e le transizioni di colore arancione sono parti di codice prodotto da noi per utilizzare il model-checker flat fornito da Yasm.

Il blocco e la transizione di colore verde caratterizzano la parte che dovrà essere implementata per produrre un model-checker gerarchico puro.

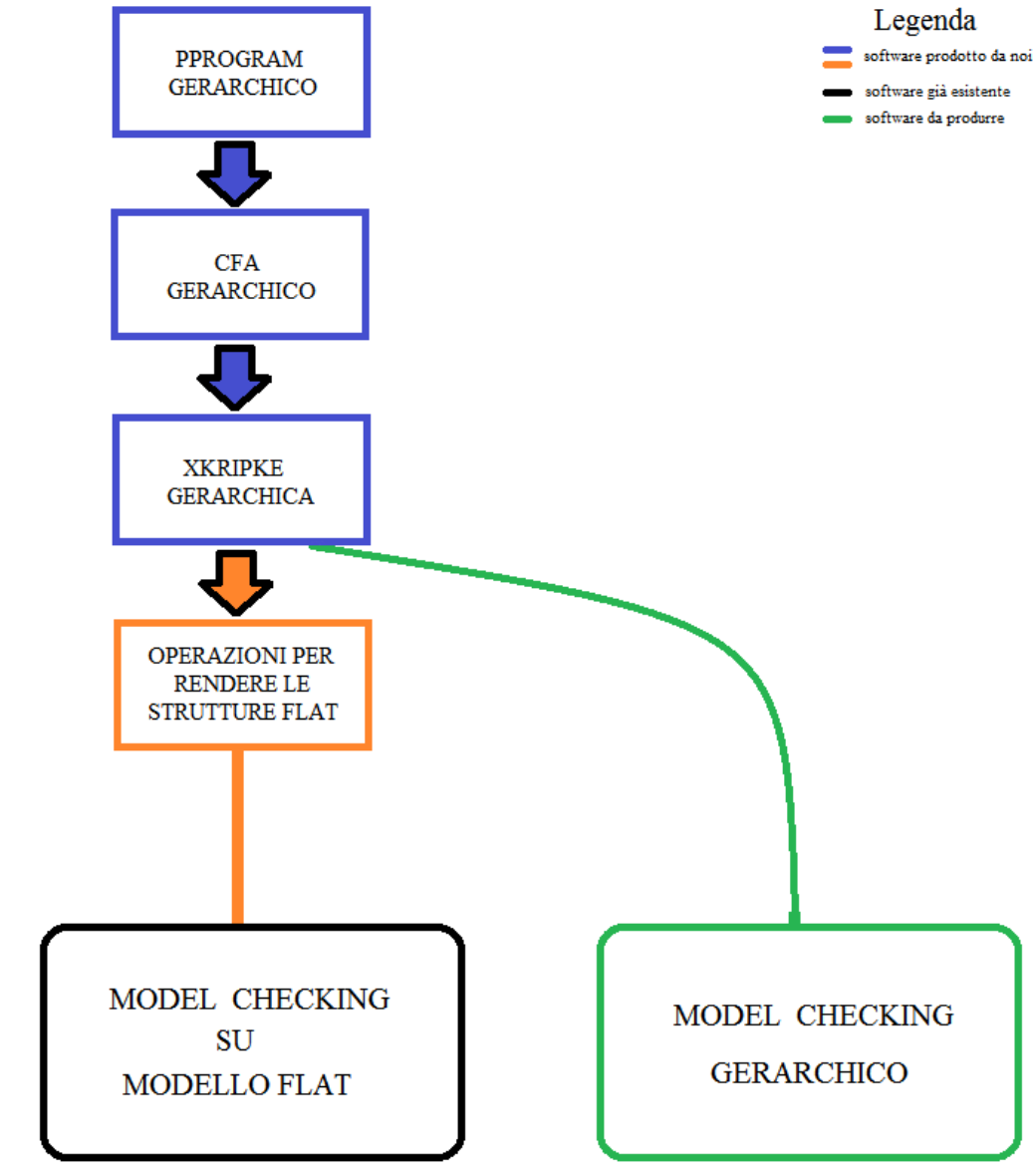
Legenda
software prodotto da noi
software già esistente
software da produrre
PPROGRAM GERARCHICO
CFA GERARCHICO
XKRIPKE GERARCHICA
OPERAZIONI PER RENDERE LE STRUTTURE FLAT
MODEL CHECKING SU MODELLO FLAT
MODEL CHECKING GERARCHICO

Figura 5-1

Bibiliografia

[1] Moshe Vardi. Alternating Automata and Program Verification.

[2] Moshe Vardi. An Automata-Theoretic Approach to Linear Temporal Logic.

[3] Orna Kupferman, Moshe Vardi, Pierre Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking.

[4] Paola Spoletini. Note su Model Checking basato su automi e Real-Time.

[5] Silvio Ghilardi. Logica Temporale Lineare.

[6] Silvio Ghilardi. Introduzione al Model-Checking Simbolico.

[7] Isabella Mastroeni. Model checking astratto.

[8] Giorgio Delzanno. Aspetti Avanzati di Rappresentazione della Conoscenza e Ragionamento Automatico.

[9] Edmund M. Clarke, Jr. Symbolic Model Checking with BDDs.

[10] Sharon Shoham, Orna Grumberg. A Game-Based Framework for CTL Counterexamples and 3-Valued Abstraction-Refinement.

[11] Rajeev Alur, Mihalis Yannakakis. Model Checking of Hierarchical State Machines.

[12] Arie Gurfinkel, Ou Wei, Marsha Chechik. Yasm: A Software Model-Checker for Verification and Refutation.

[13] Arie Gurfinkel, Marsha Chechik. Why Waste a Perfectly Good Abstraction?

[14] Kelvin Ku. Software Model-Checking: Benchmarking and Techniques for Buffer Overflow Analysis.

[15] Marsha Chechik, Arie Gurfinkel, Benet Devereux, Albert Lai, Steve Easterbrook. Data Structures for Symbolic Multi-Valued Model-Checking.

[16] Gianluigi Roveda. CENNI DI LOGICA MATEMATICA.

[17] Fabio Patrizi. Metodi Formali per il Software e i Servizi.

[18] Paolo Mazzoni. Model Checking Tutorial.

[19] Somesh Jha. The Model Checker SMV.

[20] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar e Jakob Lichtenberg. The Static Driver Verifier Research Platform.

[21] Alessandro Armando, Massimo Benerecetti, Dario Carotenuto, Jacopo Mantovani, Pasquale Spica. The Eureka Tool for Software Model Checking.

[22] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar. The software model checker BLAST.

[23] Marsha Chechik, Arie Gurfinkel, Benet Devereux, Albert Lai e Steve Easterbrook. Data Structures for Symbolic Multi-Valued Model-Checking.

[24] Benjamin Aminof, Orna Kupferman e Aniello Murano. Improved Model Checking of Hierarchical Systems.