LTL Model Checking in Prolog

Manuel Bartolini - Luca Guerra

Indice

1 Model Checking

Il Model Checking è basato sull'idea che verificare proprietà di logica temporale (proprietà LTL) su un programma a stati finiti equivalga a valutare la formula del modello visto come una interpretazione temporale.

Gli algoritmi per farlo sono abbastanza efficienti, dato che la loro complessità temporale è una funzione lineare della grandezza del programma.

Il MC può essere visto come una analisi aumentata di raggiungibilità; l'algoritmo di MC usa le specifiche di logica temporale per guidare la ricerca nello spazio degli stati del nostro protocollo cercando di dimostrare che il protocollo soddisfa le sue specifiche funzionali.

Il MC comunque risolve uno dei limiti della analisi di raggiungibilità: l'incapacità di verificare automaticamente specifiche funzionali.

D'altra parte, il MC soffre dello stesso problema che fondamentalmente ha sempre avuto l'approccio stile *reachability-analysis*: la capacità di esplorare solo spazi di dimensioni limitate. Questo problema, chiamato problema della esplosione degli stati, è la limitazione basilare più grande di entrambi gli approcci. Tutto ciò è stato argomento di una ricerca molto vasta in entrambi i contesti. Al giorno d'oggi esistono molte tecniche per il MC, ma l'approccio di base fondamentalmente è sempre lo stesso.

Il modello viene espresso come un sistema di transizioni, cioè grafo orientato formato da nodi (o vertici) e archi. Ad ogni nodo sono associate una serie di proposizioni atomiche. I nodi rappresentano gli stati del sistema, gli archi rappresentano le possibili esecuzioni che alterano lo stato, mentre le proposizioni atomiche rappresentano le proprietà fondamentali che caratterizzano il sistema in quel punto dell' esecuzione.

2 Verifica di logica temporale tramite l'automa Büchi

Il problema di Model Checking che considereremo è il seguente. Dato un programma P descritto come la esecuzione concorrente di un sistema di transizioni a stati finiti P_i e una formula LTL f, controllare tutte le infinite computazioni di P che soddisfano f.
Per risolvere il problema, seguiamo questi steps:

1. Costruiamo l'automa finito sulle infinite parole per la negazione della formula f (si usa la negata perchè porta ad un algoritmo più efficiente). L'automa risultante è $A_{\neg f}$ (la formula negata si ottiene facilmente mettendo l'operatore di negazione prima di f).

2. Eseguiamo la computazione del comportamento globale del programma P. Questo viene fatto computando una forma di prodotto ΠP_i delle transizioni dei sistemi P_i.

3. Si fa il prodotto del programma P e dell'automa $A_{\neg f}$.

4. Si verifica la *emptyness* dell'automa prodotto.

L'approccio che abbiamo appena sottolineato può essere implementato in modo da avere una maggiore efficienza rispetto agli altri approcci per il MC.

L'idea è di calcolare il prodotto P dei processi P_i e il prodotto di P con $A_{\neg f}$ nello stesso step. Questo può portare a una miglior efficienza per molte ragioni.

Per prima cosa, il prodotto di P e $A_{\neg f}$ accetta solo sequenze che non soddisfano il requisito. Ci si aspetta che ce ne siano poche (nessuna se il programma è corretto). Può essere possibile che il prodotto di P e $A_{\neg f}$ abbia meno stato raggiungibili di P. Inoltre durante la costruzione di $P \times A_{\neg f}$, non è necessario immagazzinare tutto il diagramma degli stati. E' sufficiente tenere abbastanza informazioni per controllare che la condizione 4 sia soddisfatta.

I vantaggi di ridurre il MC ad un problema di raggiungibilità sono importanti anche puramente per i problemi di safety. In questi casi è sufficiente controllare che alcuni stati siano raggiungibili.

Un'automa di Büchi è formato da una tupla $A = (\Sigma, S, \rho, s_0, F)$, dove:

- Σ è un alfabeto,
- S è un insieme di stati,
- $\rho : S \text{ x } \Sigma \rightarrow 2^S$ è una funzione di transizione non deterministica
- $s_0 \in S$ è lo stato iniziale,
- $F \subseteq S$ è un insieme di stati designati.

Un esecuzione di A su una parola infinita $w = a_1 a_2 ....$ è una sequenza infinita $s_0, s_1 ...$, dove s_0 è lo stato iniziale e $s_i \in \rho(s_{i-1}, a_i)$, per tutti gli $i \geq 1$.

Un esecuzione $s_0, s_1 ...$ è *accettante* se si trova uno stato che si ripete infinite volte, cioè per alcuni $s \in F$ ci sono infiniti i tali per cui $s_i = s$. La parola infinita w è *accettata* da A se c'è un esecuzione accettante di A su w. L'insieme numerabile di word accettate da A è denotato da $L(A)$.

Dalla definizione di automa di Büchi, è relativamente facile capire che Büchi non è *empty* se c'è qualche stato $f \in F$ raggiungibile dallo stato iniziale che è anche raggiungibile partendo da se stesso (in uno o più step). In termini teorici utilizzando i grafi, significa che il grafo rappresentante l'automa ha un ciclo raggiungibile che contiene almeno uno stato in F.
Per formalizzare questo approccio noi consideriamo processi $P_i, 1 \leq i \leq n$, che sono un sistema di transizioni a stati finiti che consiste in:

- uno spazio degli stati V_i,

- un insieme di azioni Σ_i,
- una funzione di transizione non deterministica $\sigma_i : V_i \times \Sigma_i \rightarrow 2^{V_i}$,
- uno stato iniziale $v_{0_i} \in V_i$.

Il programma P che corrisponde all'esecuzione concorrente dei processi P_i è ottenuto facendo il prodotto del sistema delle transizioni P_i. In maniera più specifica, il sistema di transizioni $P = \prod P_i$ è $P = (V, \Sigma, \sigma, V_0)$, dove

- $V = \prod V_i$, lo spazio degli stati è il prodotto cartesiano dello spazio degli stati dei P_i;
- $\Sigma = \cup\Sigma_i$, l'insieme delle azioni è l'unione degli insiemi delle azioni dei componenti;
- σ è definita da $(v'_1, ..., v'_n) \in \sigma((v_1, , ..., v_n), a)$ se
  - $v'_i \in \sigma(v_i, a)$ per ogni i tale che $a \in \Sigma_i$,
  - $v'_i = v_i$ per ogni i tale che $a \notin \Sigma_i$, es. azioni comuni a diversi processi sono sincronizzate dove altre azioni sono alternate;
- $v_0 = (v_{0_1}, ..., v_{0_n})$

Il nostro obiettivo è verificare proprietà sugli infiniti possibili comportamenti di P. Questi vengono definiti guardando P come un'automa di Büchi limitato nel quale l'insieme degli stati designati è l'intero insieme degli stati V. Da notare come la procedure di verifica che stiamo descrivendo non considera i comportamenti finiti di P (proprietà dei sistemi finiti, come ad esempio l'assenza di *deadlock*, devono essere controllate a parte).

Per controllare che P soddisfi una proprietà LTL f, dobbiamo prima costruire un'automa di Büchi basato sull'alfabeto del nostro programma $\sum$ che accetti tutte le infinite *words* che non soddisfano f.

Facciamo si che $A_{\neg f} = (\Sigma, S, \rho, s_0, F)$ sia il nostro automa di Büchi, costruito partendo dalla negazione della formula f. Il prossimo passo nelle procedura di verifica sarà calcolare $P \times A_{\neg f}$ che equivale all'automa con

- insieme degli stati $V \times S$,
- funzione di transizione $\tau : V \times S \times \Sigma \rightarrow 2^{V \times S}$ definita da $(v_2, s_2 \in \tau((v_1, s_1), a)$ se $v_2 \in \sigma(v_1, a)$ e $s_2 \in \rho(s_1, a)$,
- lo stato iniziale (v_0, s_0),

- l'insieme degli stati designati $V \times F$.

L'automa così prodotto accetta tutti i run che equivalgono a tutti i possibili comportamenti di P (accettati dall'automa P) e che violano la formula f (sono accettati dall'automa $A_{\neg f}$). In questo modo abbiamo ridotto il problema di provare che il programma P soddisfi la formula f ad un problema di verifica della *emptyness* sull'automa di Büchi $P \times A_{\neg f}$.

3 Algoritmo

Per prima cosa consideriamo l'algoritmo di ricerca standard *depth-first* che implementa la generazione del sistema di transizioni labeled F, partendo da una specifica del nostro sistema concorrente.

Dobbiamo inoltre considerare come questa ricerca possa essere estesa in modo da generare il prodotto sincrono $F \times G$, dove G è un automa Büchi che codifica una formula LTL, e per rilevare l'esistenza di cicli di accettazione in quel prodotto.

Due insiemi di stati vengono quindi inizializzati con lo stato iniziale predefinito s_0: lo *Statespace* e lo *Stack*. La ricerca inizia con una chiamata alla routine di ricerca del depth-first, Dfs(), con il parametro 1.
L'importanza di questo parametro diverrà chiara più avanti.

```
1 start_search(s0)
2  {
3    enter s0 into StateSpace;
4    push s0 onto Stack;
5    Dfs(1);
6 }
```

L'algoritmo seguente mostra invece l'espansione degli stati dei processi, nella routine dfs(). In assenza dell'automa di Büchi entrambe le chiamate di riga 5 e 16 possono essere interpretate come chiamate a dfs(N).

```
7 dfs(N) {
8    s = top(Stack);
9    for each sequential process i {
10    next = all transition in F enabled in s with Pid(t) = i;
```

```
11    for all t in next {
12      s' = successor of s after t;
13      if { s' , N} NOT into StateSpace {
14        enter {s' , N} into StateSpace;
15        push s' onto Stack;
16        Dfs(N);
17  } } }
18 pop s from Stack;
19 }
```

Nella versione generale dell'algoritmo di verifica, tuttavia, le chiamate alle righe 5 e 16 invocano la routine mostrata qui di seguito, che implementa lo step di espansione per le transizioni nell'automa di Büchi. Lo stato dell'automa di Büchi fa parte del sistema di stati composto s.

Poiché le transizioni nell'automa di Büchi rappresentano proposizioni booleane create a partire da una formula LTL sottostante, una transizione $t \in T_G$ nell'automa di Büchi G sarà abilitata se e solo se la proposizione t è valida. La sincronizzazione dei sistema F e G si ottiene alternando le chiamate a Dfs(N), sulla linea 16, e dfs(N) sulla linea 28. Ogni coppia di chiamate successive, esplora una transizione sincrona di $F \times G$.

```
20 Dfs(N) {
21    s = top(Stack);
22    next = all transition in G enabled in s ;
23    for all t in next {
24      s' = successor of s after t;
25      if { s' , N} NOT into StateSpace {
26        enter {s' , N} into StateSpace;
27        push s' onto Stack;
28        dfs(N);
29    } }
30 pop s from Stack;
31 }
```

Il codice precedente mostra solo lo step base di espansione senza la parte necessaria per rilevare la presenza dei cicli di accettazione nel sistema prodotto dal nostro sistema concorrente e dall'automa di Büchi. Per permettere anche la rilevazione dei cicli di accettazione, dobbiamo controllare,

per ogni *acceptance state*, raggiungibile in G se quello stato è raggiungibile anche partendo da se stesso (ovvero se il sistema torna ad un certo punto a transitare in questo stato). Questo viene fatto facendo partire , sempre da quello stato, una seconda ricerca depth-first in uno *Statespace* separato.

Due valori distinti per il parametro N servono per indicare in quale parte della ricerca l'algoritmo si sta muovendo. L'avvio della seconda ricerca è mostrato nelle quattro linee extra tra le linee 28 e 29 del codice mostrato qua sotto.
Se lo stato seme (ovvero lo stato accettante) è raggiungibile partendo da lui stesso questo viene rilevato e segnalato alla riga 24.

```
20 Dfs(N) {
21    s = top(Stack);
22    next = all transition in G enabled in s ;
23    for all t in next {
24      s' = successor of s after t;
24a     if N == 2 and s' == seed {
24b        report acceptace cycle;
24c        return;
24d     }
25      if { s' , N} NOT into StateSpace {
26        enter {s' , N} into StateSpace;
27        push s' onto Stack;
28        dfs(N);
28a       if N == 1 and s is an accepting state in G {
28b         seed = s ;
28c         dfs(2);
28d       }
29    } }
30 pop s from Stack;
31 }
```

Se anche esistono più esempi di cicli di accettazione l'algoritmo ne genera soltanto uno. Non c'è garanzia di generare tutti i cicli. Se, invece, l'automa di Büchi viene utilizzato per controllare se si verifica un comportamento non desiderato, cioè la violazione di un requisito di correttezza, l'esistenza o l'assenza di cicli di accettazione che soddisfano la richiesta è una prova sempre sufficiente per avere un risultato conclusivo della nostra verifica. Si

noti che quando viene scoperta l'esistenza di un ciclo di accettazione, il cammino completo che porta al ciclo è contenuto nello Stack e può essere generato come contro-esempio (counter-example) rispetto alla proprietà di correttezza che si era voluta verificare.

4 Descrizione del modello

Il linguaggio utilizzato è frutto di molte valutazioni in fase di analisi. Inizialmente si era pensato di procedere prendendo come esempio il linguaggio Promela, utilizzato dal model checker Spim. Proseguendo per questa strada però, ci si è resi conto che le ore richieste per elaborare ed implementare un linguaggio di questo tipo sarebbero state eccessive e ciò sarebbe andato a scapito del resto del progetto.

Contemporanemente si faceva invece strada l'idea che sarebbe stato più semplice e, alla fine dei conti, più produttivo implementare un programma basato sul linguaggio Prolog; per due semplici motivi.

Primo, l'obiettivo del progetto è implementare un model checker basato su Prolog, quindi probabilmente meno problemi dal punto di vista del merge tra il codice a basso livello e quello a livello utente. Secondo, puntando su Prolog avevamo già a disposizione un tool che ci sarebbe stato sicuramente molto utile, ovvero tuProlog.

Secondo, puntando su Prolog si aveva già a disposizione un tool che sarebbe stato sicuramente molto utile, ovvero tuProlog. In questo modo, basando la sintassi del linguaggio su quella Prolog si è stati in grado di utilizzare l'IDE di tuProlog fornendo all'applicazione il codice in forma di libreria. Inoltre ci si è potuti avvalere di tutte le feature offerte da 2P, come il check della teoria che torna utile in molti casi. Partendo da questi presupposti si è cominciato a pensare a quale linguaggio inventare che fosse abbastanza *user-friendly* per l'utente e allo stesso tempo permettesse di sfruttare un linguaggio come Prolog. L'idea è stata quella di unire la forza di un linguaggio dichiarativo come Prolog a basso livello con la facilità ad alto livello offerta dai linguaggi di programmazione cosiddetti imperativi. Oltre a questo poi, si doveva fornire all'utente un modo per ridurre il suo applicativo ad una macchina a stati, perciò si è deciso di strutturare i processi in questo modo:

```
nomeStato(azione, lista dei prossimi stati*).

*questa è sempre una lista, anche se lo stato seguente
è unico sarà comunque una lista con un solo elemento.
```

Facendo in questo modo si utilizza la sintassi Prolog e si permette all'utente di dividere i propri processi in una serie di stati che svolgono delle operazioni (generalmente ad ogni stato dovrebbe corrispondere una sola azione ma si è deciso di permettere l'attuazione di più istruzioni per stato).

In questo modo l'utente è in grado di scrivere il suo programma come una macchina a stati e utilizzare le funzioni fornite per permettere al programma di mettere in atto le varie operazioni richieste.

La lista completa delle funzioni comprese nella libreria è fornita nel manuale allegato, ognuna corredata dalle indicazioni di utilizzo.

A questo punto però si poneva un problema, l'utente doveva per forza essere in grado di interagire con l'*environment* ma d'altra parte il model checker aveva la necessità di legare gli stati dell'automa prodotto allo stato dell'environment in quel momento. Questo perché durante l'esplorazione dell'albero degli stati, quando il MC si accorge che una strada è da scartare torna indietro allo stato (nodo) da cui era partita l'esplorazione (per nozioni sul metodo di esplorazione utilizzato dall'algoritmo si rimanda al capitolo 3 e al capitolo 5) e riparte da lì. Per farlo naturalmente ha bisogno di ripristinare lo stato dell'environment di quel nodo perché durante l'esplorazione della strada scartata l'environment è stato soggetto a del *side effect* causato dalle varie operazioni compiute. Quindi c'era bisogno a basso livello di associare ad ogni stato l'environment ma allo stesso tempo non aveva senso far scrivere all'utente delle funzioni che come argomenti avessero anche le nozioni di environment in quello stato e di environment futuro.

Era chiaro quindi come servisse implementare una parte di libreria che si occupasse del *merge* tra le funzioni lato utente e quelle lato model checker.

Le funzioni a livello utente appaiono quindi così:

```
newvar(Name,Value)

wait(SemaphoreName)

ifeq(VariableName,Value)
```

mentre in realtà il mapping interno le fa corrispondere a queste:

```
newvar(Name,Value,Env,NEnv):- drop(var(Name,_),Env,NE),
                              append(NE,[var(Name,Value)],NEnv).

wait(PC,SemName,Env,NEnv,N,Next):- member(sem(SemName,Value,Blocked),Env),
                                  (Value > 0,!,Nvalue is Value-1,Next=N,
```

```
                        drop(sem(SemName,Value,Blocked),Env,NE),
                              append(NE,[sem(SemName,Nvalue,Blocked)],NEnv);
                              drop(sem(SemName,Value,Blocked),Env,NE),
                              append(Blocked,[PC],PCBlocked),
                              append(NE,[sem(SemName,0,PCBlocked)],NEnv),
                              atom_chars(PC,StringName),
                              append(StringName,['_','b','l','o','c','k'],NStringName)
                              atom_chars(NPC,NStringName),Next=[NPC]).

ifeq(Name,Val,Env,Env) :- member(var(Name,Val),Env),!.
```

Questo mapping interno è svolto da uno *stateSolver* che conosce semplicemente il Program Counter corrente e tramite questo va a cercare nella teoria qual'è lo stato da elaborare, ricavandone poi le azioni da svolgere. Ecco una parte del codice dello *stateSolver*:

```
stateSolver(P,Env,NEnv,[Next]) :- Fun =.. [P,atomic(Actions),Next],call(Fun),!,
                                  exec(Actions,Env,NEnv).

stateSolver(P,Env,Env,[Next]) :- Fun =.. [P,ifeq(N,V),[Then,Else]],call(Fun),!,
                                 (exec([ifeq(N,V)],Env,Env),!,Next = Then;
                                 Next = Else).

stateSolver(P,Env,Env,Next) :- Fun =.. [P,await(Cond),N],call(Fun),!,
                               execAwait(P,Cond,Env,N,Next).

stateSolver(P,Env,NEnv,Next) :- Fun =.. [P,signal(Sem),N],call(Fun),!,
                                signal(Sem,Env,Env1,PCBlock),
                                getUp(PCBlock,Env1,NEnv,N1),
                                append(N,N1,Next).

stateSolver(P,Env,NEnv,Next) :- Fun =.. [P,wait(Sem),N], call(Fun),!,
                                wait(P,Sem,Env,NEnv,N,Next).

stateSolver(P,Env,NEnv,[Next]) :- Fun =.. [P,[],N], call(Fun),!,
                                  exec([],Env,NEnv),
                                  member(Next,N).

stateSolver(P,Env,NEnv,[Next]) :- Fun =.. [P,Actions,N], call(Fun),
```

```
                                    exec([Actions],Env,NEnv),
                                    member(Next,N).
```

Dovendo fare model checking di programmi concorrenti è stato naturale fornire l'utente di alcune funzioni per gestire la sincronizzazione, abbiamo quindi introdotto la possibilità di gestire semafori. Come si può vedere dagli esempi precedenti abbiamo la funzione *wait*, ma anche la *signal* e la *await*. Una volta che un processo rimane bloccato in seguito ad una *wait* prende questa nomenclatura:

```
nomeStatoProcesso_block
```

in questo modo non corrisponde a nessuno stato scritto dall'utente e di conseguenza non viene mai eseguito.

Oltre a questo è inoltre possibile interagire con delle liste, che possono comprendere insiemi eterogenei di elementi, tutte le funzioni per utilizzarle al meglio sono spiegate nel dettaglio all'interno del manuale, qui di seguito riportiamo solo un piccolo esempio:

```
newlist(Name,List)

newlist(Name,L,Env,NEnv):- drop(list(Name,_),Env,NE),
                           append(NE,[list(Name,L)],NEnv).

addtolist(Element,ListName)

addtolist(Var,ListName,Env,NEnv):- member(list(ListName,List),Env),!,
                                   append(List,[Var],NL),
                                   newlist(ListName,NL,Env,NEnv).
```

5 Algoritmo in Prolog

Come discusso ampiamente in precedenza il *model checker* consiste in una ricerca mirata nello spazio degli stati per dimostrare la veridicità di un modello rispetto ad una formula LTL. L'obiettivo di questo progetto è l'implementazione dell'algoritmo di capitolo 3 in Prolog. Utilizzando un linguaggio dichiarativo, per implementare la verifica LTL su modelli, si può sfruttura la pecularietà del motore Prolog, il quale per soddisfare un goal genera lo spazio degli stati della soluzione grazie ad un sistema di *depth first search* e *backtracking*.

5.1 Dalla formula LTL all'automa di Büchi

Sviluppando l'algoritmo in ambiente *tuProlog* la trasformazione da una formula LTL al rispettivo automa di Büchi è stata realizzata tramite l'utilizzo di una libreria scritta in java [3] che data in ingresso la formula, asserisce nella teoria Prolog il relativo automa di Büchi.

```
?- ltl2buchi(' ! [] p ',X).
yes.
X / [init(b0),
    state(b0,accept(false)),
    trans(b0,[true],b0),
    trans(b0,[prop(p,false)],b1),
    state(b1,accept(true)),
    trans(b1,[true],b1)]

Solution: ltl2buchi('![]p',[init(b0),state(b0,accept(false)),
  trans(b0,[true],b0),trans(b0,[prop(p,false)],b1),
  state(b1,accept(true)),trans(b1,[true],b1)]))
```

Come si evince dall'unificazione della variabile X l'automa di Büchi è rappresentato da una lista, che oltre a contenere l'indicazione delle stato iniziale del sistema, contiene tanti serie di fatti che definiscono gli stati e una serie di fatti che rappresentato le transizione dell'automa.

```
state(StateName, IsAcceptance).

trans(From,Guard,To).
```

5.2 Automa prodotto

Per ottenere il prodotto tra l'automa del modello e quello di Büchi sono state implementate due regole Prolog, le quali, ricevuto in ingresso lo stato attuale del sistema restituiscono tutte le possibili transizioni. In pratica queste due regole implementano due *depth first search* una per il modello l'altra per l'automa di Büchi, che intervallandosi vicendevolmente vanno a generare una *nested depth first search*.
Ogni stato dell'automa prodotto è definito nella seguente maniera:

```
state(sys([ListPC]),buchi(BuchiPC),Env).
```

dove :

- *ListPC*: contiene la lista del *program counter* attivi nel dato istante τ;
- *BuchiPC*: è il program counter all'istante τ;
- *Env*: contiene le *environment* allo stato attuale del sistema.

Il codice sotto mostra l'implementazione della *nested depth first search*. La regola *dfsBuchi* prende in ingresso la lista degli attuali program counter e per ognuno, considerando l'*environment* attuale, genera tutte le possibili transizioni. Ogni cammino generato diventerà effettivo se e solo se chiamando la regola *dfsBuchi* con lo stesso *environment* di sopra, anche l'automa di Büchi transiterà in un nuovo stato. Se ci sono transizioni anche nella *dfsBuchi* allora viene chiamata nuovamente la regola *dfsSys*.

La ricorsione per ogni cammino generato termina quando questo cerca di esplorare uno stato già conosciuto.

In questo modo si ottiene il prodotto $F \times G$, tra il modello F e l'automa di Büchi G.

```
dfsSys(SysPC,BuchiPC,Env) :-
    sysTransition(SysPC,Env,NEnv,Next),  % executes the current system states
                                         %  and opens the all possibles paths
    enableState(Next,NEnv,NNEnv),  % sets true to the currents propositional states

    dfsBuchi(Next,BuchiPC,SysPC,Env,NNEnv).  % computes the
                                       %    automata product

dfsBuchi(SysPC,BuchiPC,CurrSys,Env,NEnv) :-
    buchiTransition(BuchiPC,Env,Accept,Next),  % executes the current system
                                         %   states and opens the all possible paths
    NextState = state(sys(SysPC),buchi(Next),NEnv),
    (
        append2StateSpace(NextState),!,
        dfsSys(SysPC,Next,NEnv)
        ;
        termination(Stack)
    ).
```

```
sysTransition(SysPC,Env,NEnv,NextPCs) :-  % executes the system state and open a
                                          % path for each program counter
    member2(PC,SysPC,RestSysPC),  % selects one program counter,
    resolver(PC,RestSysPC,Env,NEnv,NextPCs). % resolves the program counter

buchiTransition(PC,Env,Accept,Next) :-   % browse the buchi automata
    buchiAutomata(B),  % retrieves the buchi automata
    member(state(PC,Accept),B),! ,  % retrives the buchi state detail
    member(trans(PC,Trans,Next),B),  % retrives the state's transition
    transition(Trans,Env).

resolver(PC,PCList,Env,NEnv,Next) :-   % standard resolver
   stateSolver(PC,Env,NEnv,TNext),
   append(TNext,PCList,Next).
```

Particolare attenzione, è da porre sulla regola *resolver* che viene chiamata dal *sysTransition*. Questa regola grazie allo *stateSolver* del capitolo 4 permette il *binding* tra l'algoritmo di verifica e il modello del sistema lato utente.

5.3 Ricerca corse critiche

Disponendo ora dell'automa prodotto non resta che implementare la parte dell'algoritmo che permetterà di verificare se un modello soddisfa una formula LTL o meno.

Come afferma l'algoritmo di verifica di capito 3, oltre allo spazio degli stati è necessario disporre di altre 3 variabili:

- *Stack* tiene traccia dello sviluppo di un determinato cammino e nel caso di non veridicità della formula LTL verrà restituito come *counter example*;
- *N* è una etichetta che definisce il tipo di esplorazione:
  - *1* semplice;
  - *2* il cammino è giunto in uno stato seme e l'obiettivo di questa ramificazione è dimostrare la veridicità della formula LTL (negata);
- *Seed* è l'eventuale stato accettante dell'automa prodotto che diventa il seme della ricerca della veridicità

```
dfsSys(SysPC,BuchiPC,Env,Stack,N,Seed)

dfsBuchi(SysPC,BuchiPC,Env,NEnv,Stack,N,Seed)
```

In linea con l'algoritmo di capitolo 3 è stato necessario aggiungere altre due nuove regole *dfsBuchi* più prioritarie della precedente che vanno ad implementare le due condizioni presenti della funzione *Dfs(N)*.

```
dfsBuchi(SysPC,BuchiPC,CurrSys,Env,NEnv,[state(CurrSys,Env)|T],2,seed(SysPC1,Next,Env
                                % searchs a race condition
    not(stop(true)),  % stop if a race condition found yet.
    same(SysPC,SysPC1),
    same(NEnv,Env1),
    buchiTransition(BuchiPC,Env,A,Next),!,% executes the current system states
    raceCondition(T).

raceCondition([]) :- assert(stop(true)).
```

Questa regola viene eseguita se e solo se tutte le condizioni sotto elencate sono vere:

- la ramificazione corrente ha lo scopo di dimostrare la veridicità della formula (N = 2);
- la lista dei program counter in ingresso equivale a quelli presenti nel seme (*same(SysPC,SysPC1)*);
- il nuovo environment generato dal cammino equivale a quello del seme (*same(NEnv,Env1)*);
- e se l'automa di Büchi transita nello stato equivalente al seme (*Next*);

nel qual caso tutte le condizioni di cui sopra siano vere si è dimostrato che un cammino dell'automa prodotto passato da uno stato accettante diventantato *seed* si richiude in se stesso ripassando dallo stato seme. Se si è dimostrata la veridicità della formula LTL negata, allora il goal è soddisfatto e verrà restituito in uscita il contro esempio presente nello Stack che dimostra la non soddisfacibilità del modello rispetto alla formula LTL data

in ingresso.

Se una delle condizioni precedenti non è verificata allora il motore Prolog esegue la dfsBuchi di priorità immediatamente inferiore alla precedente:

```
dfsBuchi(SysPC,BuchiPC,CurrSys,Env,NEnv,[state(CurrSys,Env)|T],1,Seed) :-
    not(stop(true)),  % stop if a race condition found yet.
    buchiTransition(BuchiPC,Env,accept(true),Next),!,  % executes the current
                            % buchi states and opens the all possible paths
                            % if and only if the buchi state is acceptance.
    NextState = state(sys(SysPC),buchi(Next),1,Env),
    append2StateSpace(NextState),  % save the product state
    createSeed(CurrSys,BuchiPC,Env,NewSeed),
    expandAcceptanceState(SysPC,Next,NEnv,T,NewSeed).  % opens 2 paths

expandAcceptanceState(SysPC,BuchiPC,Env,Stack,_) :-
    dfsSys(SysPC,BuchiPC,Env,Stack,1,null).  % calls a normal dfs
expandAcceptanceState(SysPC,BuchiPC,Env,Stack,Seed) :-
    dfsSys(SysPC,BuchiPC,Env,Stack,2,Seed).  % calls a dfs that
                                % will search a race condition
```

La regola sopra viene eseguita se e solo se le condizioni sotto sono rispettate:

- la ramificazione corrente è una normale depth first search (N = 1) ;
- se lo stato *BuchiPC* è accettante;

se queste condizioni vengono soddisfatte, significa che il ramo corrente è transitato da uno stato accettante dell'automa prodotto, quindi come dice l'algoritmo di capitolo 3 il *path* corrente viene sdoppiato in due path parelleli, in cui uno dei quali continuerà una normale esplorazione dell'automa, mentre l'altro avrà l'obiettivo di dimostrare la veridicità della formula LTL (negata).

Se nessuna delle regole di cui sopra soddisfa una delle condizioni presentate viene eseguita la regola *dfsBuchi* mostrata in precedenza in occasione della realizzazione dell'automa prodotto.

5.4 Risoluzione livelock

Come discusso alla fine del capitolo 2, l'algoritmo non riesce a controllare proprietà di sistemi finiti, quindi per notificare situazioni di *livelock* e *deadlock* sono state necessarie delle modifiche.

Si consideri il possibile caso di figura 1 in cui un cammino non abbia più modo di proseguire perché si verifica una condizione di *livelock*.
L'algoritmo di model checking una volta approdato in uno stato prima di transitare nel successivo controlla, tramite lo *StateSpace*, che questo non sia già stato esplorato, nel caso di figura la transizione riporta sempre nello stesso stato quindi l'algoritmo una volta raggiunto il nodo pota tutte gli archi uscenti terminando così la propria esecuzione.

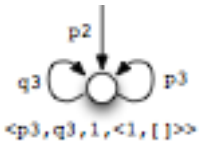

Figura 1: livelock state

Per ovviare a questo problema ogni volta che la transizione di uno stato consiste in un ciclo su se stesso(*await*), viene generato una copia dello stato attuale in modo che la transizione sia fatta su quest'ultimo. In questo modo, se lo stato soggetto a livelock è accettante, grazie al nuovo stato temporaneo si permette all'algoritmo di model checking di compiere il ciclo per dimostrazione l'eventuale veridicità della formula LTL (negata).

Qui sotto sono riportate le regole che risolvono il problema del livelock.

```
resolver(PC,PCList,Env,NEnv,Next) :-  % resolves a possible trick state
                                      % with a deadlock pc
    isBlock(PC),
    trickState(PCList),!,
    member2(NPC,PCList,RestList),
    stateSolver(NPC,Env,NEnv,TNext),
    append([PC],TNext,N),
    append(N,RestList,Next).

resolver(PC,PCList,Env,NEnv,Next) :-  % resolves a cycle state
```

```
    isTemp(PC),!,
    member2(NPC,PCList,RestList),
    stateSolver(NPC,Env,NEnv,N),
    livelockResolver(PC,N,TNext),
    append(TNext,RestList,Next).

livelockResolver(PC,N,Next) :-  % manages livelock cycle state
    notTrickState(N),!,
    getRightState(PC,RPC),
    append([RPC],N,Next).

livelockResolver(PC,N,Next) :-  % manages livelock cycle state
    append([PC],N,Next).
```

6 Valutazione performance

In questo progetto è stato implementato uno dei primi o forse il primo algoritmo di model checking proposto da *Gerard J. Holzmann*, senza porsi problemi di ottimizzazione. Non essendo implementata nessuna tecnica di ottimizzazione, all'aumentare dei processi l'aumento degli stati ha una crescita esponenziale che si riflette nel tempo di computazione richiesto dal programma per dimostrare una data proprietà LTL. È stato utilizzato come primo benchmark l'algoritmo dei *readers and writers* controllando la proprietà di mutua esclusione:

```
 init([r1,w1],[newvar(nr,0),newsem(mutexR,1),newsem(rw,1)]).

%%READER
r1(wait(mutexR),[r2]).
r2(ifeq(nr,0),[r3,r4]).
r3(wait(rw),[r4]).
r4(incvar(nr,1),[r5]).
r5(signal(mutexR),[r6]).
r6([],[r7]). %%reading
r7(wait(mutexR),[r8]).
r8(decvar(nr,1),[r9]).
r9(ifeq(nr,0),[r10,r11]).
r10(signal(rw),[r11]).
```

```
r11(signal(mutexR),[r1]).

%%WRITER
w1(wait(rw),[w2]).
w2([],[w3]). %%writing
w3(signal(rw),[w1]).

?- modelChecker(' [] ! ( r6 /\ w2)',X).
```

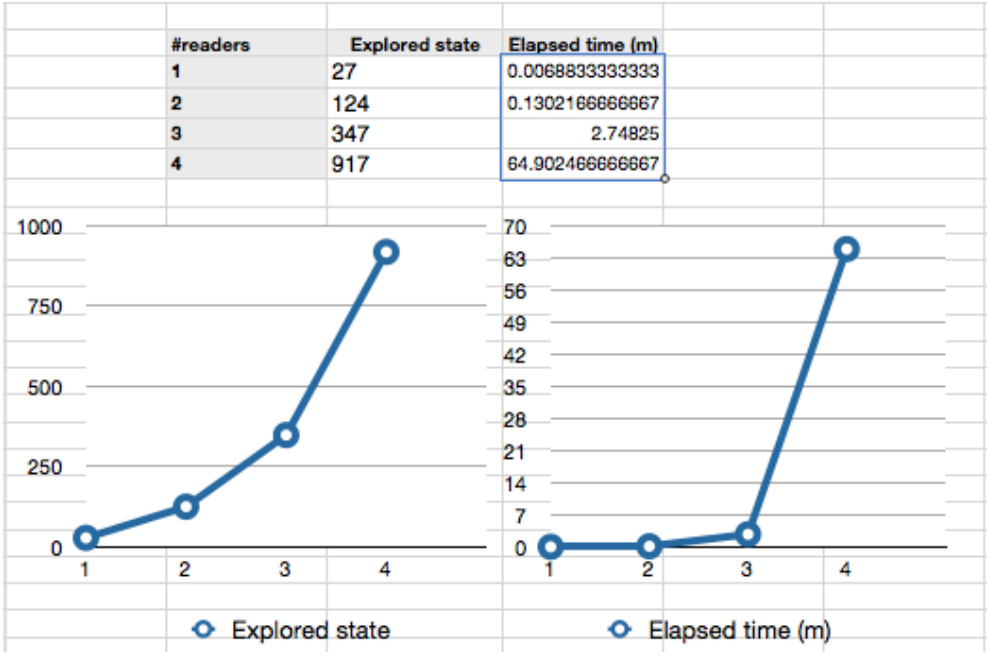

Figura 2: readers and writers

Come mostrano le tabelle e i grafici di figura 2 all'aumentare dei readers gli stati così come il tempo computazionale aumenta in maniera esponenziale. Il motivo di un gradino così elevato tra 3 e 4 readers è dovuto al controllo fatto per ogni transizione della *nested depth first search*.

```
append2StateSpace(State) :-   statespace(L),
```

```
not(exists(State,L)),
retract(statespace(L)),!,
append(L,[State],R),
assert(statespace(R)).
```

La regola sopra viene chiamata per ogni transizione generata dalla *dfsBuchi*. Questa, prima di inserire lo stato futuro nella lista controlla se questo è già presente, quindi il tempo di ricerca aumenta linearmente all'aumentare degli stati presenti.

Per cercare di migliorare i tempi di verifica si è ricorsi ad un semplice accorgimento partendo da questa premessa:

la ricerca nello spazio degli stati è una *depth first search* e quindi un cammino difficilmente si chiude su stesso nella radice.

Si è cambiato il principio di inserimento degli stati futuri invece che in coda alla lista in testa. Questo come mostrano i dati di figura 3 ha portato ad un abbassamento dei tempi di verifica.

#readers	Explored state	Elapsed time (m)
1	27	0.0040166666667
2	124	0.1070833333333
3	347	2.2114833333333
4	917	47.8499

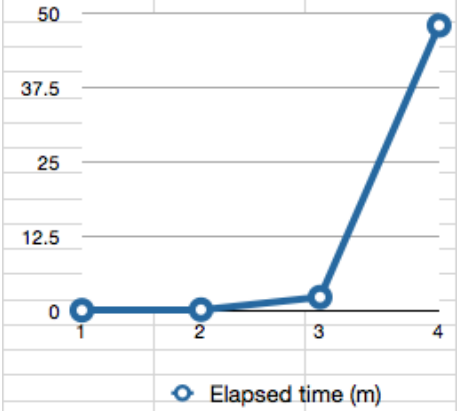

Figura 3: readers and writer

Allo stato attuale non ci sono altri metodi per diminuire i tempi di esplorazione di tutto l'automa prodotto, si può però ricorrere ad un altro accorgimento per abbassare i tempi nel caso in cui una proprietà non sia verificata. La modifica in questione è semplicemente cambiare il principio di ricerca da *depth first search* a *breadth search* in questo modo esplorando lo spazio degli stati in ampiezza invece di controllare un cammino completo per volta, si controllano tanti cammini in contemporanea. La prova del miglioramento è data dalle tabelle di figura 4, in cui si cerca di dimostrare la mutua esclusione dei *Dining Philosophers* in cui però il model checker termina prematuramente dopo aver trovato una situazione di deadlock.

```
init([a1,b1,c1],[newsem(fork1,1),newsem(fork2,1),newsem(fork3,1)]).


a1([],[a2]). % think
a2(wait(fork3),[a3]).
a3(wait(fork1),[a4]).
a4([],[a5]). %eat
a5(signal(fork3),[a6]).
a6(signal(fork1),[a1]).

b1([],[b2]). % think
b2(wait(fork1),[b3]).
b3(wait(fork2),[b4]).
b4([],[b5]). %eat
b5(signal(fork1),[b6]).
b6(signal(fork2),[b1]).

c1([],[c2]). % think
c2(wait(fork2),[c3]).
c3(wait(fork3),[c4]).
c4([],[c5]). %eat
c5(signal(fork2),[c6]).
c6(signal(fork3),[c1]).
```

Depth first search					
#philosopher	Stat time (ms)	End time (ms)	Elapsed time (ms)	Explored state	Elapsed time (m)
3	1.32843E+12	1.32843E+12	458	35	0.0076333333333
5	1.32847E+12	1.32848E+12	2054093	766	34.234883333333
Breath search					
#philosopher	Stat time (ms)	End time (ms)	Elapsed time (ms)	Explored state	Elapsed time (m)
3	1.32846E+12	1.32846E+12	183	8	0.00305
5	1.32846E+12	1.32846E+12	465	14	0.00775
7	1.32846E+12	1.32846E+12	414	20	0.0069
9	1.32846E+12	1.32846E+12	505	26	0.0084166666667
11	1.32846E+12	1.32846E+12	986	32	0.0164333333333

Figura 4: Confronto tra il metodo di ricerca *depth first search* e *breadth search*

7 Conclusioni e sviluppi futuri

Ai fini del progetto e del monte ore associate si è implementato un model checker che permettesse la verifica di formule LTL. A nostro avviso il sistema è completo ma può essere migliorato cercando di velocizzare il *checking* e ampliando il linguaggio fornito.

Come discusso nel capitolo precedente le performance non sono state un punto principale del progetto. Per aumentare quindi le performance si potrebbe come spiegato in [1] sfruttare una riduzione statica dello spazio degli stati.

Un altro miglioramento potrebbe essere la gestione del nome gli stati, infatti per semplicità allo stato attuale l'utente definisce il nome di uno stato come un letterale ed un numero.

Riferimenti bibliografici

[1] Gerard J. Holzmann, *An Improvement in Formal Verification*, 1994

[2] Giuseppe Oliva, *Automi di Büchi etichettati e Model Checking LTL*, 2006

[3] Dimitra Giannakopoulou, Flavio Lerda, *Efficient translation of LTL formulae into Büchi automata* , 2001

[4] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, *Memory-Efficent Algorithms for the Verification of Temporal Proprietes*, 1992

[5] Gerard J. Holzmann, Doron Peled, Mihalis Yannakakis, *On Nested Depth First Search*, 1991