Università degli Studi di Roma "Sapienza"

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesina di Metodi Formali per l'Ingegneria del Software

Logiche Temporali al servizio dell'Ingegneria del Software: CTL, CTL*, TRIO e LTLB

Giacomo Bernini
sotto la supervisione del Prof. Toni Mancini

Anno Accademico 2008/2009

INDICE

❖ 1. Introduzione

Durante il corso di Metodi Formali per l'Ingegneria del Software è stato illustrato l'utilizzo della logica temporale LTL per verificare proprietà di documenti di analisi, come diagrammi degli stati UML, e di programmi, per verificarne la terminazione e la correttezza parziale o totale.

Il presente lavoro si prefigge lo scopo di scendere in profondità nello studio delle logiche temporali, vagliando diverse possibilità e analizzandone i pregi e i difetti, confrontandoli con la già conosciuta LTL, e introducendo per ognuna di esse possibili applicazioni nell'ambito del Model Checking.

Nel capitolo 2 ci si avvicinerà alla logica CTL, per certi versi simile a LTL, e alla logica CTL*, che costituisce un sovrainsieme di entrambe le precedenti, e verrano mostrate applicazioni di CTL nel capitolo 3.

Nel capitolo 4 verrà introdotta la logica TRIO, e verrà descritta una visione generale del complesso ma interessante processo per ottenerne un'applicazione pratica. Si vedrà come TRIO abbia importanti analogie con la logica LTLB, ovvero la logica LTL con operatori per il passato, che viene pertanto descritta nel conclusivo capitolo 5, insieme ad una sua applicazione pratica.

❖ 2. CTL e CTL*

2.1 Sintassi e Semantica di CTL

La logica temporale aggiunge alla logica proposizionale o del primo ordine la capacità di gestire proposizioni relative a tempi diversi.

Permette quindi di esprimere proprietà legate al tempo (come vedremo in seguito fa uso di operatori temporali).

Esistono due principali famiglie distinte in base al modo di modellare il tempo:

- Logiche temporali lineari (*Linear Time*)
- Logiche temporali ramificate (*Branching Time*)

La logica CTL (*Computational tree logic*) è una *branching-time logic*: ovvero, il suo modello di tempo è una struttura ad albero in cui ogni istante ha un passato unico ma un futuro non determinato; è quindi possibile esprimere formule logiche su cammini (ossia sequenze di transizioni di stato) a partire da un determinato stato iniziale. La caratteristica principale di questa logica è che ogni formula deve essere premessa da un quantificatore di cammino, per cui ogni formula viene sempre riferita all'insieme dei cammini a partire da uno stato iniziale.

La logica CTL usa proposizioni atomiche come elementi costitutivi per marcare gli stati di un sistema. Quindi combina queste proposizioni in formule utilizzando gli operatori logici e temporali.

Gli *operatori logici* utilizzati, comuni alla maggior parte delle logiche (anche non temporali), sono i ben conosciuti:

- ¬,∧,∨,=> e <=>.

La logica CTL è caratterizzata inoltre dai seguenti *operatori temporali*:

- X p (*next*) : una certa proprietà *p* si verifica nell'istante successivo,
- F p *(eventually)* : *p* si verifica in futuro
- G p *(always)* : *p* è sempre verificata
- p U q (*until*) : *p* precede *q* o, in altre parole, *p* è vera prima che sia vera *q*.

Questi operatori non possono però essere utilizzati liberamente: essi devono essere sempre prefissi da un quantificatore di cammino, *A* oppure *E*:

- A p (*All*): *A p* è vera se per tutti i cammini vale *p* (necessità).
- E p (*Exists*): *E p* è vera se esiste almeno un cammino in cui vale *p* (eventualità).

La sintassi di CTL è definita come segue:

- Ogni proposizione atomica è una formula CTL.
- Se p e q sono formule CTL, allora lo sono anche: ¬ p, (pq), AX p, EX p, A(p U q), E(p U q).

Dalle precedenti formule possono essere ricavate tutte le altre costruibili a partire dagli operatori visti in precedenza. Ad esempio, la formula *Fp* può essere scritta come *trueU(p).*

Esempi di formule esprimibili con CTL sono riportati in figura 1.

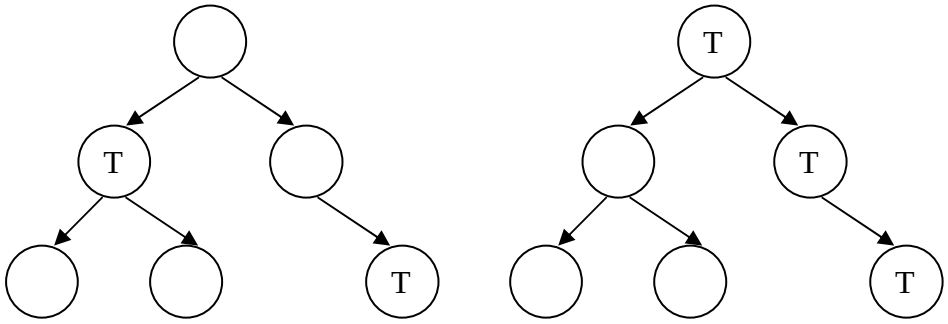

Figura 1: Esempi di proprietà espresse con CTL

La prima, *AFp*, asserisce che per ciascun cammino della struttura riportata in figura, in un qualche stato del cammino stesso si verifica la proprietà *p*, mentre la seconda, *EGp*, asserisce che esiste un cammino per il quale la proprietà *p è* sempre verificata: nel primo caso, infatti, tutti i cammini a partire dalla radice contengono almeno uno stato contrassegnato con una *T* che sta a indicare che *p è* vera in quello stato, nel secondo esiste un cammino dalla radice alle foglie in cui *p è* vera in tutti gli stati.

Come già osservato, le formule espresse con CTL si riferiscono sempre ad un cammino oppure alla totalità dei cammini. Come avremo modo di capire meglio più avanti, è questa la caratteristica più importante di CTL, nel bene e nel male.

La semantica è definita usando il concetto di *struttura di Kripke*, K = (S, R, L), dove:

- S è l'insieme degli stati in cui il sistema evolve
- R è la relazione di stato successivo
- L è una funzione che dato uno stato fornisce l'insieme delle proposizioni atomiche vere in tale stato.

Il cammino di un modello K = (S,R,L) è un'infinita sequenza di stati (s_0, s_1, s_2...) in cui ogni coppia di stati (s_i, s_{i+1}) è un elemento di R.

La notazione K, s |= f significa che la formula f è vera nello stato s del modello di Kripke K. L'interpretazione di una formula *f* CTL rispetto a un modello di Kripke K è riportato di seguito:

- K, s |= p iff p ∈ L(s) (for p ∈ S).
- K, s |= ¬j iff K, s |≠ j.
- K, s |= j∧y iff K, s |= j and K , s |= y.
- K, s |= j∨y iff K, s |= j or K, s |= y.
- K, s |= j⇒y iff if K, s |= j then K, s |= y.
- K, s |= ⊤.
- K, s |≠ ⊥.
- K, s_i |= AXp iff per ogni cammino (s_i, s_{i+1}, ...) vale K, s_{i+1} |= p.
- K, s_i |= EXp iff esiste un cammino (s_i, s_{i+1}, ...) tale che K, s_{i+1} |= p.
- K, s_i |= AGp iff per ogni cammino (s_i, s_{i+1}, ...) vale ∀ j ≥ i K, s_j |= p.
- K, s_i |= EGp iff esiste un cammino (s_i, s_{i+1}, ...) tale che ∀ j ≥ i K, s_j |= p.
- K, s_i |= AFp iff per ogni cammino (s_i, s_{i+1}, ...) vale ∃ j ≥ i K, s_j |= p.
- K, s_i |= EFp iff esiste un cammino (s_i, s_{i+1}, ...) tale che ∃ j ≥ i K, s_j |= p.
- K, s_i |= A(pUq) iff per ogni cammino (s_i, s_{i+1}, ...) vale ∃ j ≥ i K, s_j |= q AND ∀ i ≤ k < j : K, s_k |= p.
- K, s_i |= E(pUq) iff esiste un cammino (s_i, s_{i+1}, ...) tale che ∃ j ≥ i K, s_j |= q AND ∀ i ≤ k < j : K, s_k |= p.

2.2 CTL vs LTL

La logica LTL consente di predicare formule senza alcuna considerazione dei cammini effettuati dal sistema nella sua evoluzione dallo stato iniziale; questo significa che gli operatori temporali utilizzati con LTL non devono essere prefissi da un quantificatore di cammino come in CTL.

La logica LTL è pertanto definita dagli operatori

- *X, F, G, U, R*

senza quantificatori.

Gli operatori possono essere composti e quindi è possibile definire formule come

- *GFp*

ossia *p è* vera infinitamente spesso, o

- *FGp*

p è sempre vera a partire da un certo istante futuro.

In questo caso, diversamente da quello precedente, non si considera un albero di possibili esecuzioni ma solo l'insieme $S = (s_0, s_1, \ldots, s_n)$ degli stati raggiungibili dal modello. Con LTL è dunque possibile considerare proprietà in un certo senso più precise, ossia che descrivono non strutture (cammini o sequenze di stati) ma stati: con LTL è possibile affermare che una certa formula sia vera a partire da un certo stato in poi oppure che sia vera infinitamente spesso rispetto ad un insieme di stati, non un insieme di cammini.

Le due logiche hanno caratteristiche differenti, ma il loro potere espressivo non è comparabile; in linea teorica, quindi, non è possibile affermare che una sia migliore dell'altra. Quello che possiamo dire è che esistono proprietà esprimibili in CTL ma non in LTL e viceversa: per esempio sappiamo che non esiste nessuna formula CTL in grado di esprimere la formula LTL *FGp* e non esiste nessuna formula LTL in grado di esprimere la formula CTL *AG*(*EFp*).

Inoltre LTL viene spesso preferita rispetto a CTL per la caratteristica di poter definire proprietà degli stati senza vincoli sui cammini, come spiegato sopra; la proprietà menzionata prima *FGp*, è importante in diversi contesti e non è esprimibile in CTL. Ma se da un lato LTL offre la possibilità di esprimere in un certo senso proprietà più interessanti, dall'altro presenta problemi in termini di complessità dell'algoritmo di verifica. Considerata una struttura di Kripke $K = (S,R,L)$, per quanto riguarda gli algoritmi di verifica di una formula logica *f* su *K* sappiamo che:

- Esiste un algoritmo per determinare se una formula CTL *f* è vera in uno stato *s* della struttura *K* che termina in un tempo pari a $O(|f| \cdot |S| + |R|)$.

- Esiste un algoritmo per determinare se una formula LTL *f* è vera in uno stato *s* della struttura *K* che termina in un tempo pari a $|K| \cdot 2^{O(|f|)}$.

E' da notare come LTL sia esponenziale rispetto alla dimensione della formula, ossia al numero di stati con cui questa viene rappresentata, per cui si potrebbe concludere che il problema di LTL risieda solo nelle formule. In realtà LTL è lineare rispetto a *K*, non rispetto a *S* come CTL: in effetti la struttura *K* utilizzata per la verifica formale, nella rappresentazione esplicita in cui si elencano tutti i possibili

valori di verità delle variabili, assume dimensione esponenziale rispetto agli stati. Pertanto LTL risulta esponenziale sia rispetto alla formula che allo spazio degli stati.

*2.3 CTL**

CTL* è una logica che combina il potere espressivo di CTL con quello di LTL. In altri termini, gli operatori temporali visti sin qui possono essere utilizzati in CTL* senza alcun vincolo o costrizione.

La sintassi di CTL* è definita come segue:

- Ogni proposizione atomica è una formula CTL*.
- Se p è una proposizione atomica, allora chiamiamo *State Formula* una formula CTL* del seguente tipo: p, $\top$, $\bot$, $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, A α, E α, dove φ e ψ sono *State Formulas* e α è una *Path Formula.*
- Chiamiamo invece Path Formula una formula CTL* del seguente tipo: φ, $\neg\alpha$, $\alpha \wedge \beta$, $\alpha \vee \beta$, X α, G α, F α, α U β, dove φ è una *State Formula* e α e β sono *Path Formulas.*

Una *State Formula* viene valutata sugli stati, una *Path Formula* sui cammini.

Anche con questa logica la semantica è definita a partire dal concetto di Struttura di Kripke. La notazione K, s |= f significa che la formula f è vera nello stato s del modello di Kripke K. L'interpretazione di una formula *f* CTL* (di tipo *State Formula*) rispetto a un modello di Kripke K è riportato di seguito:

- $K, s \models \neg\varphi$ iff $K, s \not\models \varphi$.
- $K, s \models \varphi \wedge \psi$ iff $K, s \models \varphi$ and $K, s \models \psi$.
- $K, s \models \varphi \vee \psi$ iff $K, s \models \varphi$ or $K, s \models \psi$.
- $K, s \models$ E α iff esiste un cammino $\pi(s_i, s_{i+1}, ...)$ tale che $K, \pi \models \alpha$.
- $K, s \models$ A α iff per ogni cammino $\pi(s_i, s_{i+1}, ...)$ vale $K, \pi \models \alpha$.

L'interpretazione di una formula *f* CTL* (di tipo *Path Formula*) rispetto a un modello di Kripke K è riportato di seguito:

- $K, \pi \models \varphi$ iff $K, s_0 \models \varphi$.
- $K, \pi \models \neg\alpha$ iff $K, \pi \models \alpha$.
- $K, \pi \models \alpha \wedge \beta$ iff $K, \pi \models \alpha$ and $K, \pi \models \beta$.
- $K, \pi \models \alpha \vee \beta$ iff $K, \pi \models \alpha$ or $K, \pi \models \beta$.
- $K, \pi \models$ F α iff $\exists\, i \geq 0$ tale che $K, \pi^i \models \alpha$.
- $K, \pi \models$ A α iff $\forall\, i \geq 0$ vale $K, \pi^i \models \alpha$.

- $K, \pi \models X\ \alpha$ iff $K, \pi^1 \models \alpha$.
- $K, \pi \models \alpha\ U\ \beta$ iff $\exists\, i \geq 0$ tale che $K, \pi^i \models \beta$ and $\forall\ j.(0 \leq j \leq i)$ vale $K, \pi^j \models \alpha$.

Sia CTL che LTL sono sottoinsiemi di CTL*. La loro relazione è sintetizzata graficamente in figura 2.

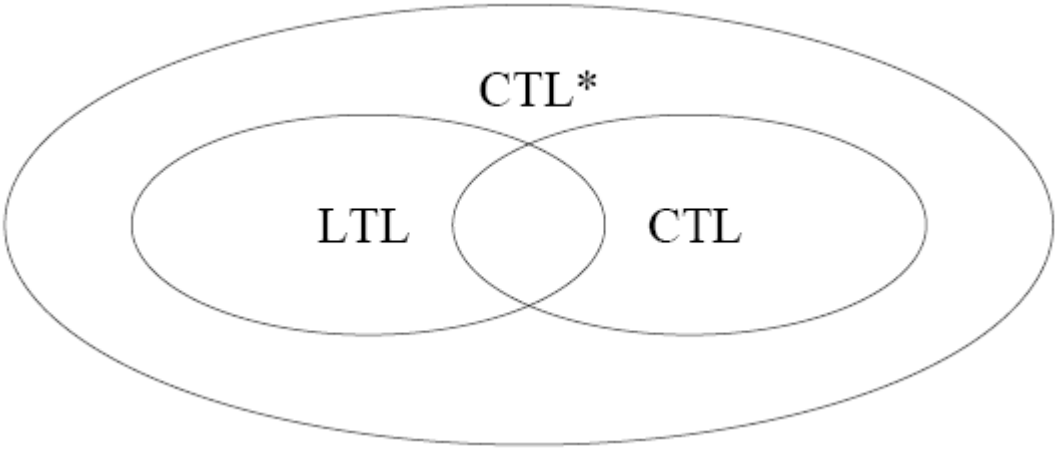

Figura 2: Relazione fra LTL, CTL e CTL.*

Una formula in CTL rimane invariata se scritta in CTL*:

- φ in CTL ⇒ φ in CTL* (per esempio, AG (p ⇒ EFq).

Una formula in LTL va prefissa con il quantificatore di cammino A se scritta in CTL*:

- φ in LTL ⇒ A φ in CTL* (per esempio, A(GFp ⇒ GFq)).

Dalla figura precedente si può evincere facilmente come esistano formule di CTL* che non possono essere scritte nè in LTL nè in CTL. Ad esempio:

- E(GFp ⇒ GFq).

Per concludere la panoramica su CTL*, si può analizzare la sua complessità computazionale, mettendola in relazione con quella delle altre due logiche viste sin qui. La figura 3 ci mostra come la complessità computazionale di CTL* per il Model Checking sia dello stesso ordine di LTL.

Logic	Complexity w.r.t. $\lvert \varphi \rvert$	Complexity w.r.t. $\lvert \mathcal{M} \rvert$
LTL	PSpace-Complete	P (linear)
CTL	P-Complete	P (linear)
CTL*	PSpace-Complete	P (linear)

Figura 3: Confronto fra LTL, CTL e CTL secondo la complessità computazionale del Model Checking.*

Si può notare come tutte e tre le logiche risultino lineari rispetto alla struttura M (anche se solo CTL è lineare anche rispetto a S, il numero degli stati), mentre CTL è l'unica a mostrare una complessità P-Complete rispetto alla formula.

❖ 3. Model Checking utilizzando CTL

3.1 Cenni generali sul Model Checking

La progettazione di sistemi software sempre più complessi ha visto recentemente lo sviluppo di diversi approcci alla progettazione stessa. Tra di essi, un approccio diffuso è quello guidato dalla definizione di un modello del sistema considerato utilizzando notazioni formali (Automi a Stati Finiti, logiche) o semi formali (UML) per valutare proprietà del sistema prima della sua effettiva implementazione.

A supporto di tale attività, il *model checking* è una tecnica di verifica automatica che consente di definire un modello del sistema con una notazione formale, quindi specificare le proprietà desiderate in modo rigoroso tramite logiche temporali e infine verificare tali proprietà allo scopo di cercare di dimostrarne la validità o di identificare errori nelle scelte progettuali.

Per poter introdurre la verifica automatica nel ciclo progettuale sono necessari i seguenti passi:

- Formalizzazione dei requisiti del sistema da verificare
- Modellazione del sistema da verificare
- Esecuzione, cioè verifica che il sistema dato soddisfi i requisiti
- Integrazione della verifica automatica nel ciclo progettuale

Formalizzazione dei requisiti del sistema da verificare

I requisiti del sistema da verificare possono essere definiti usando vari formalismi. Tipici esempi sono UML, MSC (*Message Sequence Charts*), logiche temporali (es. CTL, LTL, etc).

In molti casi, ma non in tutti, la proprietà da verificare può essere definita come un insieme di stati indesiderati. Quello che si vuole è che il sistema nella sua evoluzione non raggiunga mai uno stato indesiderato. In questi casi l'attività di verifica automatica consiste nell’analisi di raggiungibilità, ovvero calcolare gli insiemi di stati che il sistema può raggiungere nel corso della sua evoluzione e quindi verificare che nessuno di questi stati raggiungibili sia uno stato indesiderato.

Modellazione del sistema da verificare

Il sistema da verificare può essere descritto usando linguaggi orientati alla definizione dell'implementazione del sistema oppure usando linguaggi orientati alla definizione dei requisiti di un sistema.

Di seguito alcuni esempi di linguaggi orientati alla definizione dei requisiti di un sistema:

- Finite State Machines. Ad esempio come in SDL, Statecharts, etc.
- Timed Automata.
- Term Rewriting.
- Process Algebras (es. CCS).
- Logics (e.g. Temporal Logics).

Tali modelli vengono tipicamente usati per validare le specifiche informali del sistema stesso.

Esecuzione della Verifica

Concettualmente la verifica automatica corrisponde ad un testing esaustivo del sistema da verificare.

Ovviamente eseguire un testing esaustivo non è possibile a causa dell'enorme numero di *test cases* che sarebbero necessari.

I model checkers ottengono lo stesso risultato utilizzando opportuni algoritmi e strutture dati. L'efficienza di tali algoritmi è di fatto il fattore che ha permesso la transizione del model checking dal mondo accademico al mondo industriale.

Alcuni degli approcci usati di frequente sono:

- Model Checking Esplicito: viene usata una *hash table* per memorizzare gli stati visitati.
- Model Checking Simbolico: l'insieme degli stati visitati viene rappresentato con la sua funzione caratteristica che viene a sua volta rappresentata con un OBDD (*Ordered Binary Decision Diagram*).
- Bounded Model Checking: Il problema di verifica viene trasformato in un problema di *Soddisfacibilità* su espressioni booleane (*SAT*).

Integrazione della verifica automatica nel ciclo progettuale

La verifica automatica (*model checking*) può essere facilmente introdotta nel ciclo progettuale sia del software (reattivo) che dell'hardware poichè non richiede un lungo training per essere usata.

Esistono diversi tools di verifica automatica ampiamente utilizzati, tra cui:

- SPIN
- NuSMV

- SMV
- altri...

A seconda del sistema da analizzare un model checker può essere più indicato di un altro.

3.2 Un caso di studio

Consideriamo un esempio per mostrare come vengono utilizzate le strutture di Kripke per rappresentare alcune caratteristiche dei sistemi reali: il caso considerato è quello di un forno a microonde che può avere quattro possibili configurazioni (stati), può ricevere o eseguire comandi e per il quale sono fornite le seguenti specifiche:

- Il forno si trova inizialmente in uno stato in cui non ha ricevuto nessun comando e non sta svolgendo alcuna attività; inoltre, la porta è aperta.
- Il forno può ricevere il comando *start*, ma non può avviare la cottura se la porta non viene prima chiusa.
- Se il forno riceve il comando *start* e la porta è chiusa allora può avviare la cottura, durante la quale non può ricevere altri comandi; una volta terminata la cottura, il forno ritorna nello stato iniziale e apre automaticamente la porta.

Il modello che rappresenta le specifiche suddette è riportato in figura 4. La struttura definisce il valore della funzione *L* per ogni stato, ossia il valore di verità di ogni variabile (*close*, *start*, *cooking*), l'insieme degli stati $S = (S0, S1, S2, S3)$ e l'insieme *R* delle transizioni.

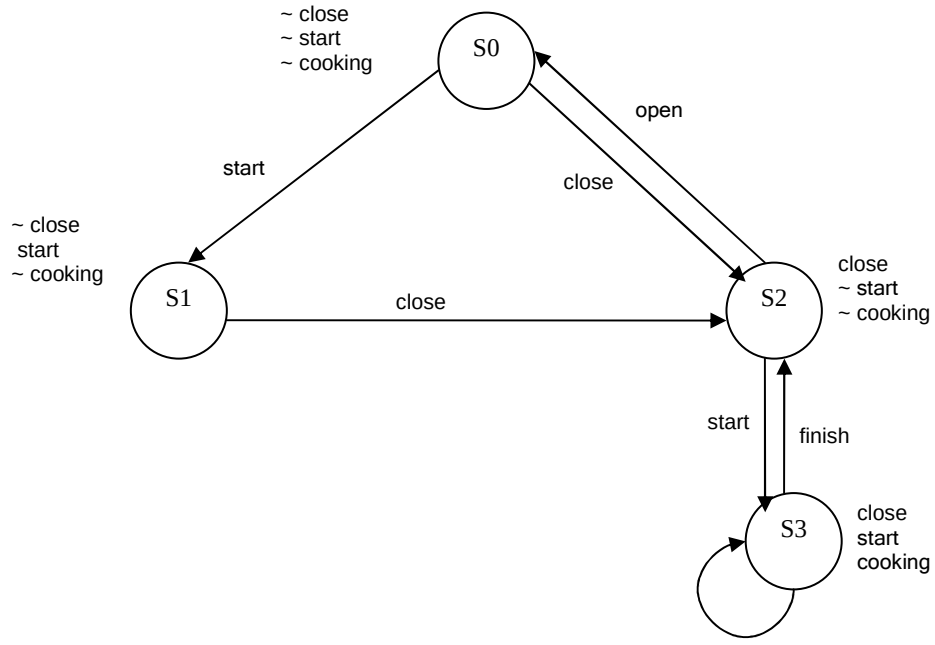

Figura 4: Il modello del forno a microonde.

Le logiche temporali consentono di esprimere proprietà relative all'evoluzione del sistema nel tempo; è da sottolineare che con il *model checking* è possibile dimostrare che un modello soddisfa una certa proprietà, ma è impossibile dimostrare che la specifica di una proprietà o di un insieme di proprietà copra l'insieme di tutte le proprietà soddisfacibili dal sistema.

Una classificazione delle proprietà di maggior interesse nell'applicazione di questo metodo di verifica può essere la seguente:

- *Safety Properties* (something bad will not happen).
- *Fairness Properties* (something is successful infinitely often).
- *Liveness Properties* (something good will happen).

Una *Safety Property* è una proprietà che esprime una condizione che deve sempre verificarsi oppure una condizione di pericolo che non si deve mai verificare per la sicurezza del sistema. Nel caso dell'esempio preso in considerazione una proprietà di *safety* potrebbe essere la seguente, utilizzando la logica CTL:

- AG(*cooking* $\Rightarrow$ A(*close* U *cooking*))

che vuole significare che deve essere sempre vero che se il forno è attivo prima deve essere stata chiusa la porta e che essa deve essere rimasta chiusa fino a quando il forno entra nella fase di cottura.

Una *Fairness Property* è una proprietà utile per definire ad esempio la politica di uno *scheduler* che deve sempre mandare in esecuzione qualsiasi processo. Una adeguata proprietà di *fairness* garantisce che tutte le richieste di esecuzione di quel processo devono essere prima o poi soddisfatte. Nel caso del forno si può pensare di formulare la seguente proprietà:

- AG AF($\neg$*cooking* $\wedge \neg$*close*)

che vuole significare che il forno non può rimanere in stato di cottura all'infinito. Al contrario, prima o poi deve ritornare nello stato iniziale definito precedentemente e per il quale valgono le condizioni specificate nella formula, ossia che il forno non è in cottura e la porta è aperta; la condizione definita in questo modo garantisce che per qualunque stato in cui si trova il forno è sempre vero che in futuro il forno si riporta nella condizione iniziale.

Una *Liveness Property* specifica qualcosa che deve accadere in futuro e viene talvolta utilizzata per identificare delle situazioni di errore. Viene qui riportata a titolo informativo anche se in realtà le proprietà di questo genere sono meno interessanti e

spesso riconducibili alle prime due tipologie. Un esempio di *liveness* riconducibile al caso di studio è la seguente formula:

AG(*start* ⇒ AF(*cooking*))

che sta a significare che se riceve il comando di avvio il forno prima o poi deve entrare in funzione. Spesso queste proprietà vengono utilizzate per scoprire errori in fase di specifica, un comportamento atteso che non si verifica.

3.3 Modellazione tramite NuSMV

NuSMV è un *model checker* simbolico sviluppato a partire da SMV (S*ymbolic Model Verifier*), un *model checker* che già permetteva la verifica di specifiche in CTL per processi cooperanti che comunicano attraverso variabili condivise: con NuSMV è stata introdotta la possibilità di specificare anche formule LTL, ed ha pertanto sostituito il suo predecessore.

La rappresentazione interna del linguaggio utilizzata per scrivere modelli è basata su OBDD (*Ordered Binary Decision Diagram*). Storicamente SMV è nato per la verifica di componenti hardware e pertanto il linguaggio stesso non contiene elementi tipici dell'interazione tra processi, non esiste il concetto di canale e nessun meccanismo di scambio di messaggi.

Tuttavia è possibile definire uno o più moduli che non possono comunicare tra loro ma possono condividere variabili globali. Riguardo al caso di studio, è stato definito un *module* che rappresenta il forno (*oven*) e come esso modifica il proprio stato in base agli stimoli esterni, come in figura 5. In questo caso gli stimoli esterni sono modellati tramite una variabile globale che può essere modificata dal modulo *main* e letta dal modulo *oven*.

```
MODULE main
VAR
        command: {close, start, open, finish};
        myOven: oven(command);
ASSIGN
        init(command):= start;

MODULE oven(com)
VAR state: {S0, S1, S2, S3};
ASSIGN
        init(state):=S0;
        next(state):=
                case
                        (state=S0) & (com=close) : S2;
                        (state=S0) & (com=start) : S1;
```

```
                    (state=S1) & (com=close) : S2;
                    (state=S2) & (com=open) : S0;
                    (state=S2) & (com=close) : S3;
                    (state=S3) & (com=finish) : S2;
                    1: state;
            esac;
    next(com):=
            case
                    state=S1 : close;
                    (state=S2) & (com=finish) : open;
                    state=S3 : finish;
                    1: com;
            esac;

SPEC
--safety
--AG(state=S3 -> A[state=S2 U state=S3])
--fairness
--AG( AF(state=S0))
--liveness
AG(com=start -> AF(state=S3))
```

Figura 5: Codice NuSMV per il caso del forno a microonde.

NuSMV non dispone di alcuna interfaccia grafica, tuttavia lo strumento fornisce alcune importanti funzionalità: è infatti possibile verificare proprietà ed effettuare simulazioni con opportuni parametri di configurazione; con NuSMV è possibile specificare proprietà sia in LTL che in CTL, la specifica delle proprietà viene inserita all'interno del modello ed è possibile predicare sia sulle variabili locali sia su quelle globali.

NuSMV è uno strumento che sta ricevendo ampio interesse per diversi aspetti:

- Permette di utilizzare entrambe le logiche CTL e LTL per definire specifiche di un modello.
- Utilizza una rappresentazione simbolica che consente di rendere più efficiente l'esecuzione degli algoritmi di verifica.
- La maggior parte delle caratteristiche non incluse nel linguaggio è rappresentabile con elementi aggiuntivi (per esempi variabili booleane condivise che possono essere scambiate tra moduli ed in caso usate come semafori).

3.4 Integrazione con specifiche CTL in casi di studio modellati tramite LTL

Utilizzando il caso di studio del forno a microonde, abbiamo visto come utilizzare CTL per definire specifiche di problemi generici. Il caso del forno, in realtà, è traducibile anche in altri linguaggi di logica temporale, come LTL. Analizzeremo ora un caso di studio modellato tramite LTL, per evidenziarne dei limiti nelle specifiche che possono essere superati tramite l'utilizzo di CTL.

In figura 6 viene riportato il diagramma di un cliente di un istituto bancario, modellato dal punto di vista di quest'ultimo. Gli eventi *paga* e *in_ritardo* sono azioni esogene alle quali l'istituto reagisce tramite cambiamenti della maniera in cui percepisce il cliente (ovvero del suo stato) e senza effettuare alcuna azione.

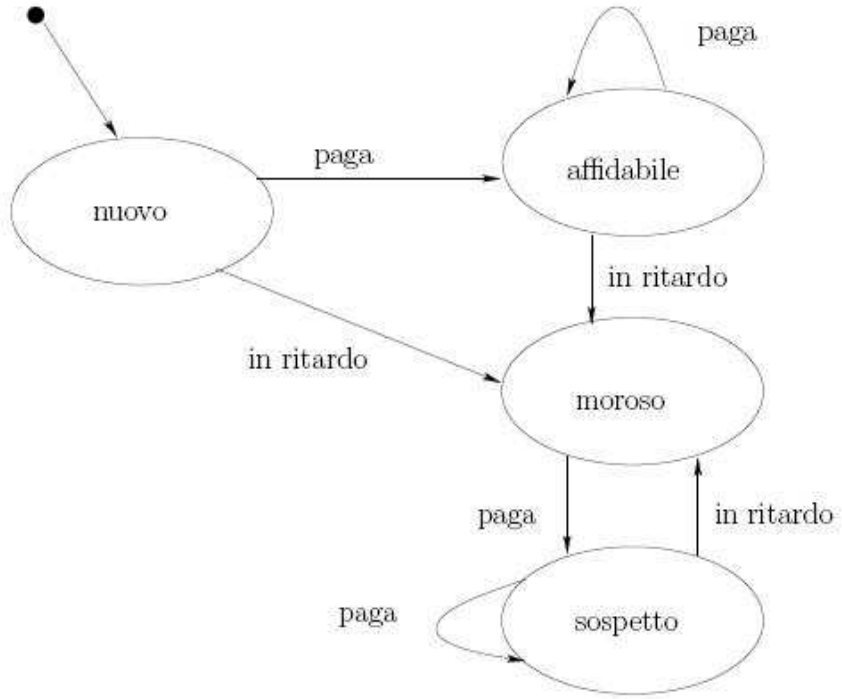

Figura 6: Diagramma UML degli stati e delle transizioni per un cliente di istituto bancario.

Utilizzando LTL è possibile definire molte specifiche sul modello in questione. Ne vengono riportate alcune:

- dopo il primo evento il cliente è affidabile o moroso: ($stato = nuovo \wedge evento \mathrel{!}= evento\ nullo \rightarrow X(stato = affidabile \vee stato = moroso)$.
- un cliente che continua a pagare rimane affidabile o nuovo: ($G\ evento \mathrel{!}= ritardo) \rightarrow G(stato = nuovo \vee stato = affidabile)$.

Ciononostante, LTL non copre l'intera gamma di specifiche che possono essere pensate sul modello del cliente. Ad esempio, non esiste alcuna specifica LTL che possa tradurre la seguente condizione:

- Un cliente può sempre diventare moroso.

Una traduzione della suddetta specifica in CTL può essere al contrario trovata:

- *AG(EF(stato = moroso)).*

Allo stesso modo, la specifica:

- Un cliente può finire in una situazione tale da non poter più diventare affidabile.

viene tradotta in CTL con:

- *EF(AG(stato != affidabile).*

Questo perchè, parlando genericamente, LTL non ha la capacità di descrivere proprietà che *possono* accadere, ma solo quelle che *inevitabilmente* accadranno (o non accadranno). In figura 7 viene riportato il codice che modella il caso di studio del cliente, corredato di specifiche CTL (che ovviamente restituiscono entrambe *true*).

```
MODULE cliente_banca
-- rappresenta un diagramma degli stati e delle transizioni
VAR
-- descrive gli stati, gli eventi e le azioni
stato : {nuovo, affidabile, moroso, sospetto};
evento: {paga, ritardo, null};
TRANS
-- descrive le transizioni
case
stato = nuovo & evento = paga: next(stato) = affidabile;
stato = nuovo & evento = ritardo: next(stato) = moroso;
stato = affidabile & evento = paga: next(stato) = affidabile;
stato = affidabile & evento = ritardo: next(stato) = moroso;
stato = moroso & evento = paga: next(stato) = sospetto;
stato = sospetto & evento = paga: next(stato) = sospetto;
stato = sospetto & evento = ritardo: next(stato) = moroso;
-- per tutti i casi non contemplati dal diagramma S&T
1: next(stato) = stato;
esac
-- fine MODULE cliente_banca

MODULE main
VAR
-- un cliente dell'oggetto "cliente_banca"
c: cliente_banca;
ASSIGN
-- assegna lo stato iniziale a c
init(c.stato) := nuovo;
-- TRANS
-- descrizione di un "generatore di eventi" (in questo caso `e superflua)
-- case
-- 1: next(c.evento) = paga | next(c.evento) = ritardo |
-- next(c.evento) = null;
-- esac
-- fine MODULE main

--LTLSPEC
```

```
    -- Alcune specifiche soddisfatte
    -- INIZIALMENTE IL CLIENTE `E NUOVO
    -- c.stato = nuovo
    -- DOPO IL PRIMO EVENTO IL CLIENTE `E AFFIDABILE O MOROSO
    -- c.stato = nuovo & c.evento != null -> X (c.stato = affidabile | c.stato = moroso)
    -- DOPO IL PRIMO EVENTO IL CLIENTE NON SAR`A MAI PI`U NUOVO
    -- (usiamo "phi U psi | G phi" al posto di "phi W psi")
    -- (c.evento = null U X G c.stato != nuovo) | G c.evento = null
    -- UN CLIENTE CHE CONTINUA A PAGARE RIMANE AFFIDABILE O NUOVO
    -- (G c.evento != ritardo) -> G (c.stato = nuovo | c.stato = affidabile)
    -- UN CLIENTE IN RITARDO ANCHE SOLO UNA VOLTA NON SAR`A MAI PI`U AFFIDABILE
    -- G(F c.evento = ritardo -> F G c.stato != affidabile)
    -- specifica migliore, usando W
    --(c.evento != ritardo U X G c.stato != affidabile) | G c.evento != ritardo

    SPEC
    -- UN CLIENTE PUO' SEMPRE DIVENTARE MOROSO
    AG(EF (c.stato = moroso))
    -- UN CLIENTE PUO' FINIRE IN UNA SITUAZIONE TALE DA NON POTER PIU' DIVENTARE
AFFIDABILE
    --EF(AG c.stato != affidabile)
```

Figura 7: Codice NuSMV del caso del cliente dell'istituto bancario con specifiche CTL.

❖ 4. TRIO

4.1 Sintassi

TRIO è un linguaggio di logica del primo ordine, arricchito con operatori temporali del passato e del futuro, che permette di esprimere proprietà il cui valore di verità può cambiare nel tempo, e fornisce mezzi e modalità per esprimere in maniera precisa e quantitativa sia la distanza in tempo tra eventi che la lunghezza di intervalli temporali.

Ogni formula TRIO assume significato rispetto ad un istante di tempo che è lasciato implicito, relativamente ad una struttura temporale usata per assegnare valori di verità a formule in istanti diversi del dominio temporale.

L'alfabeto di TRIO comprende una serie di nomi di variabili, di funzioni, di predicati e un insieme di simboli di operatori. E' importante sottolineare alcune cose, a questo proposito:

- Le variabili possono essere *locali*, il cui valore può cambiare nel tempo, o *globali*, che sono al contrario invarianti nel tempo. Ad ogni variabile viene associato un dominio, che rappresenta l'insieme dei valori che essa può assumere. Tra i domini di variabile c'è il *dominio temporale*, separato dagli altri, che deve essere di tipo numerico.
- Anche i predicati sono divisi tra quelli dipendenti dal tempo e quelli da esso indipendenti.
- Gli operatori sono divisi nei simboli proposizionali, nei quantificatori e negli operatori temporali, *Futr* e *Past*. In accordo con la sintassi definita formalmente in seguito, la formula *Futr(p,t)* (rispettivamente, *Past(p,t)*) significa che p è vero in un istante che è spostato di t unità di tempo nel futuro (rispettivamente, nel passato), rispetto all'istante di tempo lasciato implicito nella formula.

La sintassi di TRIO è definita come segue:

- Ogni proposizione atomica è una formula TRIO.
- Se p e q sono formule TRIO, x è una variabile globale e t è un termine di tipo temporale, allora sono formule TRIO anche: $\neg$ p, p $\wedge$ q, $\forall$ x p, Futr(p,t), Past(p,t).

Un gran numero di operatori temporali possono essere ottenuti a partire da *Futr* e *Past*. Di seguito ne sono elencati alcuni.

- AlwF(p) **= def** $\forall$ t (t > 0 $\wedge$ Futr (true, t) $\Rightarrow$ Futr (p, t)).

p sarà vero in ogni istante futuro.

- AlwP(p) = **def** $\forall t\,(t > 0 \wedge \text{Past}(\text{true}, t) \Rightarrow \text{Past}(p, t))$.

p è stato vero in ogni istante passato.

- SomF(p) = **def** $\neg\text{AlwF}\,(\neg p)$.

p sarà vero in almeno un istante futuro.

- SomP(p) = **def** $\neg\text{AlwP}\,(\neg p)$.

p è stato vero in almeno un istante passato.

- Sometimes(p) = **def** $\text{SomP}\,(p) \vee p \vee \text{SomF}\,(p)$.

Esiste un istante, passato o futuro, in cui p è stato o sarà vero.

- Always (p) = **def** $\text{AlwP}\,(p) \wedge p \wedge \text{AlwF}\,(p)$.

p è stato vero in ogni istante passato e sarà vero in ogni istante futuro.

- Lasts(p, t) = **def** $\forall t'(0 < t' < t \Rightarrow \text{Futr}(p, t'))$.

p sarà vero per i prossimi t istanti di tempo.

- Lasted(p, t) = **def** $\forall t'(0 < t' < t \Rightarrow \text{Past}(p, t'))$.

p è stato vero negli ultimi t istanti di tempo.

- Until (p, q) = **def** $\exists t\,(\text{Futr}\,(q, t) \wedge \text{Lasts}\,(p, t))$.

q sarà vero in un istante futuro e fino a quell'istante sarà vero p.

- Since (p, q) = **def** $\exists t\,(\text{Past}\,(q, t) \wedge \text{Lasted}\,(p, t))$.

q è stato vero in un istante passato e da quell'istante è stato vero p.

- Before (p, q) = **def** $\text{Sometimes}\,(p \wedge \text{SomF}\,(q))$.

p è stato vero o sarà vero in un istante di tempo, passato o futuro, precedente all'istante di tempo, passato o futuro, nel quale è stato vero o sarà vero q.

4.2 Semantica

La semantica di TRIO si basa sul concetto di struttura temporale, dalla quale si può derivare sia l'informazione su uno stato, ovvero un assegnamento di valori a variabili e predicati locali, sia su una funzione di valutazione che assegna ad ogni formula TRIO un distinto valore di verità per ogni istante del dominio temporale.

Una struttura temporale S = (I,T,G,L) è definita da:

- Un insieme di domini I.
- Un dominio temporale T, di tipo numerico.
- La parte di struttura indipendente dal tempo G(ξ, Π, Φ), dove ξ è una funzione di valutazione definita su un insieme di variabili globali x tale che $\xi(x) \in I_j$, per qualche $I_j \in I$, Π è una funzione di valutazione definita su un insieme di predicati globali $p_1 \ldots p_n$ tale che $\Pi(p) \subset I_{p1} \times \ldots \times I_{pn}$ e $I_j \in I$, per ogni $j \in [1 \ldots n]$, Φ è una funzione definita su un insieme di funzioni f_j tale che $\Phi(f_j)$: $I_{fj1} \times \ldots \times I_{fjn} \rightarrow I_{f0}$ e $I_{fji} \in I$ per ogni $i \in [1 \ldots n]$ e $I_{f0} \in I$.

- La parte di struttura dipendente dal tempo L, definita come $\{(\eta_i, \Pi_i)$ tale che $i \in T\}$, dove ogni coppia definisce uno stato di S, $\eta_i(y) \in I_j$, per qualche $I_j \in I$ e ogni variabile locale y, $\Pi_i(p) \subseteq I_{p1} \times \ldots \times I_{pn}$ e $I_{pj} \in I$ per ogni $j \in [1 \ldots n]$ e ogni predicato locale p.

Una struttura temporale S è detta *adeguata* per una formula TRIO se I contiene i domini di valutazione per tutte le variabili occorrenti, e se le funzioni ξ, η_i, Π, Π_i e Φ assegnano a tutte le variabili globali e locali, tutti i predicati globali e locali e tutte le funzioni valori del tipo appropriato.

Se S è una struttura temporale adeguata per una formula TRIO allora è possibile definire una funzione di valutazione che assegna un valore a tutti i termini e formule costruibili a partire da S. Chiamiamo questa funzione $S_i^{(G,L)}$ e, ad esempio, il valore di verità di una formula temporale è definito come segue:

- $S_i^{(G,L)}(Futr(p,t)) = true$ **iff** $v = S_i^{(G,L)}(t)$, con $i+v \in T$, $\bigwedge S_{i+v}^{(G,L)}(p) = true$.
- $S_i^{(G,L)}(Past(p,t)) = true$ **iff** $v = S_i^{(G,L)}(t)$, con $i-v \in T$, $\bigwedge S_{i-v}^{(G,L)}(p) = true$.

Una formula TRIO f è *temporalmente soddisfacibile* in una struttura temporale S se S è adeguata per essa ed esiste un istante di tempo $i \in T$ per cui $S_i^{(G,L)}(f) = true$. In questo caso, si può affermare che S costituisce un *modello* per f. Una formula f è *temporalmente valida* in S se $S_i^{(G,L)}(f) = true$ per ogni $i \in T$.

Una formula TRIO è temporalmente chiusa se soddisfa una delle seguenti condizioni:

- E' una formula atomica costituita da un predicato globale applicato a termini contenenti esclusivamente costanti e variabili globali.
- Ha la forma *Sometimes(p)* o *Always(p)*, dove p è una qualunque formula TRIO.
- Ha la forma $\neg p$, dove p è una formula temporalmente chiusa.
- Ha la forma $p \wedge q$, dove p e q sono formule temporalmente chiuse.
- Ha la forma $\forall x\ p$, dove x è una variabile globale e p è una formula temporalmente chiusa.

Una *specifica* TRIO è definita come una qualunque formula TRIO temporalmente chiusa.

Una formula TRIO è detta *temporalmente invariante* se, in tutte le strutture temporali adeguate, è esattamente temporalmente valida o temporalmente insoddisfacibile.

Un teorema di fondamentale importanza afferma che:

- Ogni formula TRIO temporalmente chiusa (e quindi ogni specifica scritta in logica TRIO) è temporalmente invariante.

In generale, il problema della soddisfacibilità di una formula TRIO è indecidibile. Per questo motivo viene qui di seguito introdotto un sottoinsieme decidibile di TRIO, ottenuto tramite alcune semplificazioni:

- Gli assiomi TRIO sono limitati alle sole formule mappabili in LTL con operatori del passato: *Past*, *Futr*, *Lasts*, *Lasted*, *Until* e *Since*.
- Il dominio temporale è costituito dall'insieme infinito dei numeri naturali.
- Tutti gli altri domini sono costituiti da insiemi finiti.
- Nessuna variabile temporale è permessa.

La logica LTL con aggiunta di operatori del passato (LTLB) verrà introdotta nel capitolo successivo (cap. 5). Per tradurre un assioma TRIO in formula LTL (con operatori del passato) è sufficiente ricorrere alla definizione ricorsiva della funzione di traduzione

- $\tau : F_{TRIO} \mapsto F_{LTL}$

tramite il seguente elenco di regole:

- $\tau(p) := p$.
- $\tau(\neg p) := \neg\, \tau(p)$.
- $\tau(p \wedge q) := \tau(p) \wedge \tau(q)$.
- $\tau(p \vee q) := \tau(p) \vee \tau(q)$.
- $\tau(Past(p, t)) := P^t\, \tau(p)$.
- $\tau(Futr(p, t)) := X^t\, \tau(p)$.
- Se $t > 0$ $\tau(Lasts(p, t)) := X\, \tau(p) \wedge X^2\, \tau(p) \wedge \ldots \wedge X^{t-1}\, \tau(p)$.
- Se $t = 0$ $\tau(Lasts(p, t)) := \top$.
- Se $t > 0$ $\tau(Lasted(p, t)) := P\, \tau(p) \wedge P^2\, \tau(p) \wedge \ldots \wedge P^{t-1}\, \tau(p)$.
- Se $t = 0$ $\tau(Lasted(p, t)) := \top$.
- $\tau(Until(p,q)) := \tau(p)\ U\ \tau(q)$.
- $\tau(Since(p,q)) := \tau(p)\ S\ \tau(q)$.

La logica TRIO può venire aumentata con costrutti *object-oriented* per supportare specifiche modulari, generando una nuova logica chiamata TRIO modulare, o TRIO+, di cui verrà data nelle prossime righe solo una semplice introduzione. TRIO+ mette a disposizione gli usuali concetti di *classe*, *oggetto* (chiamato *modulo*) ed *ereditarietà*, con relativa rappresentazione grafica in stile UML. Una specifica di

TRIO+ viene costruita definendo delle classi, che possono essere *semplici* o *strutturate*. Una classe semplice contiene esclusivamente un insieme di assiomi logici con dichiarazione e definizione dei predicati, delle variabili e delle funzioni che compaiono nell'interfaccia della classe. Le vere specifiche TRIO+ vengono costruite attraverso le classi strutturate, classi le cui istanze contengono moduli che sono istanze di altri classi.

4.3 Introduzione al Model Checking tramite TRIO2ProMeLa

SPIN (Simple Promela INterpreter) è un model checker sviluppato presso i Bell Laboratories. Adatto alla specifica di sistemi distribuiti concorrenti e di pubblico dominio, SPIN è oggi utilizzato da migliaia di specificatori. Direttamente creato per il software, SPIN lavora "al volo" ovvero senza bisogno di pre-costruire la struttura di Kripke per la verifica. Il modello del sistema ha un'architettura *a processi concorrenti* ed è specificato in *Process Meta Language* (ProMeLa), un linguaggio non deterministico con sintassi C-like; le proprietà, invece, vengono formalizzate per la maggior parte in LTL ma anche in forma di automi di *Büchi*.

La soluzione proposta per il passaggio da TRIO a SPIN è chiamata TRIO2ProMeLa. Questa soluzione prevede di ottenere, a partire dalle specifiche TRIO, dei processi scritti in ProMeLa, che vengono poi eseguiti in SPIN.

Una specifica TRIO consiste in un insieme di formule in logica temporale, e non contiene alcun componente operazionale (come ad esempio uno *state-transition system*) che possa generare valori. Al contrario, le formule TRIO possono solo definire vincoli sui valori che le variabili presenti nella specifica possono assumere. Pertanto, il problema di verificare proprietà in TRIO utilizzando il Model Checking prende la forma di verificare la validità di una formula logica del tipo:

- spec → prop

dove *spec* è un insieme di formule TRIO che descrivono proprietà che vengono assunte vere per il sistema analizzato, e *prop* una formula TRIO che descrive una proprietà che vogliamo verificare essere implicata da *spec*. Nell'approccio utilizzato *spec* assume un ruolo simile a quello giocato dal *modello* classico del Model Checking (come ad esempio un programma ProMeLa in SPIN).

La traduzione delle formule TRIO nell'insieme di processi ProMeLa deriva dalla correlazione fra logica temporale e *alternating automata.* In realtà, viene effettuata una duplice traduzione: prima le formule TRIO vengono tradotte in 2AMCA, quindi dal 2AMCA si passa al codice ProMeLa. 2AMCA (*2-way Alternating Modulo Counting Automata*), che non viene trattato in questa sede, è una versione dei *Büchi alternating automata*.

Il codice ProMeLa generato dalle formule TRIO attua quindi una simulazione di un *alternating automaton*, invece che di un *Büchi automaton*, facendo si che la dimensione del codice sia proporzionale alla lunghezza della formula iniziale.

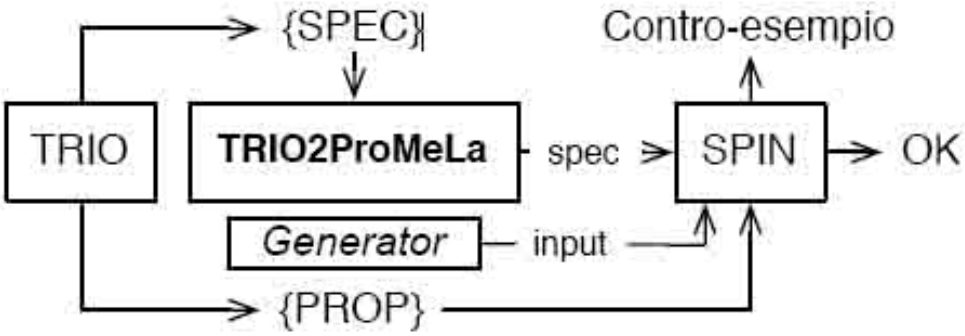

Figura 8: Descrizione ad alto livello dei componenti e del funzionamento di TRIO2ProMeLa.

I processi ProMeLa, eseguiti in SPIN, agiscono globalmente come un accettore di linguaggio, definito sull'alfabeto di *spec*. Come descritto in Figura 8, è pertanto necessario un componente che si occupi della generazione, nel tempo, di tutti i possibili input per i processi ProMeLa. Per evitare il noto problema dell'esplosione del numero degli stati, vengono attuate delle ottimizzazioni che rendono la tecnica utilizzabile in pratica, come l'utilizzo di TRIO+ e la gestione efficiente degli operatori del passato.

La semplicità della verifica totalmente automatica tramite il Model Checking si paga con una maggior complessità del meccanismo di traduzione e, soprattutto, con le limitazioni sulle formule (operatori) ammissibili negli assiomi TRIO.

❖ 5. LTLB

5.1 Sintassi e Semantica

Esistono tre logiche derivate dal concetto generale di Linear Temporal Logic:

- Linear Temporal Logic with Both past and future operators (LTLB).
- Linear Temporal Logic with only Past operators (LTLP).
- Linear Temporal Logic with only future operators (LTL).

Mentre LTL è largamente usata e conosciuta e LTLP quasi del tutto inutilizzata, recentemente l'attenzione nei confronti di LTLB è cresciuta notevolmente.

La logica LTLB è caratterizzata dai seguenti *operatori temporali*:

- X p (*next*) : una certa proprietà *p* si verifica nell'istante successivo.
- Y p *(yesterday)* : *p* si è verificata nell'istante precedente.
- p U q (*until*) : *q* si verificherà in qualche istante futuro, e fino a quell'istante sarà vera *p*.
- p S q (*since*) : *q* si è verificata in qualche istante passato, e da quell'istante è stata vera *p*.

Da questi operatori di base possono essere ottenuti tutti gli altri operatori temporali:

- F p (*future*) : *p* si verificherà in un qualche istante futuro.
- G p *(globally)* : *p* si vericherà in ogni istante futuro.
- O p (*once*) : *p* si è verificata in un qualche istante passato.
- H p (*historically*) : *p* si è verificata in ogni istante passato.

La sintassi di LTLB è definita come segue:

- Ogni proposizione atomica è una formula LTLB.
- Se p e q sono formule LTLB, allora lo sono anche: ¬ p, p∧q, pUq, pSq, Xp, Yp.

La semantica è definita a partire dall'usuale concetto di *struttura di Kripke*, K = (S, R, L). L'interpretazione di una formula *f* LTLB rispetto a un modello di Kripke K è riportato di seguito:

- K, s |= p iff p ∈ L(s) (for p ∈ S).
- K, s |= ¬p iff K, s |≠ p.

- K, s |= $p \wedge q$ iff K, s |= p and K , s |= q.
- K, s |= $\top$.
- K, s |$\neq \bot$.
- K, s_i |= Xp iff K, s_{i+1} |= p.
- K, s_i |= Yp iff K, s_{i-1} |= p.
- K, s_i |= pUq iff $\exists\ j \geq i$ K, s_j |= q AND $\forall\, i \leq k < j$: K, s_k |= p.
- K, s_i |= pSq iff $\exists\ 0 \leq j \leq i$ K, s_j |= q AND $\forall\, j \leq k < i$: K, s_k |= p.

LTLB non aggiunge potere espressivo rispetto a LTL. Qualunque formula esprimibile in LTLB è esprimibile anche in LTL. In ogni caso, LTLB viene preferita a LTL perchè permette di scrivere specifiche più semplici e più corte, rendendole più leggibili e, di conseguenza, diminuendo il numero degli errori commessi in fase di modellazione.

Aggiungere gli operatori del passato non aumenta la complessità del Model Checking, che rimane PSPACE-complete, come per LTL. Ciononostante, non è banale la questione di applicare la logica praticamente. Molti problemi sono sorti ad esempio nel Model Checking con SPIN, e ancora nessun algoritmo è stato ufficialmente implementato in maniera completa.

5.2 Integrazione con specifiche LTLB in casi di studio modellati tramite LTL

Per osservare come può essere applicata in pratica la logica LTLB, si può prendere in considerazione un qualunque caso di studio modellato tramite LTL. In effetti, viene qui preso nuovamente in considerazione il caso di studio del cliente dell'istituto bancario già analizzato in precedenza (*par. 3.4*).

Tramite LTLB possiamo scrivere qualunque proprietà esprimibile anche in LTL. Ad esempio:

- Se un cliente è affidabile, nello stato precedente era nuovo o affidabile: [*G(c.stato=affidabile -> Y (c.stato=nuovo | c.stato = affidabile))*].
- Se non si sono mai verificati eventi il cliente è nuovo: [*G(H(c.evento=null) -> c.stato=nuovo)*].
- Dopo il primo evento il cliente non sarà mai più nuovo: [*G(c.evento=null | (c.stato!=nuovo S c.evento!=null))*].

In figura viene riportato il codice che modella il caso di studio del cliente, corredato di specifiche LTLB e delle equivalenti specifiche LTL. Ovviamente, tutte e tre le specifiche ritornano *true*.

```
MODULE cliente_banca
-- rappresenta un diagramma degli stati e delle transizioni
VAR
-- descrive gli stati, gli eventi e le azioni
stato : {nuovo, affidabile, moroso, sospetto};
evento: {paga, ritardo, null};
TRANS
-- descrive le transizioni
case
stato = nuovo & evento = paga: next(stato) = affidabile;
stato = nuovo & evento = ritardo: next(stato) = moroso;
stato = affidabile & evento = paga: next(stato) = affidabile;
stato = affidabile & evento = ritardo: next(stato) = moroso;
stato = moroso & evento = paga: next(stato) = sospetto;
stato = sospetto & evento = paga: next(stato) = sospetto;
stato = sospetto & evento = ritardo: next(stato) = moroso;
-- per tutti i casi non contemplati dal diagramma S&T
1: next(stato) = stato;
esac
-- fine MODULE cliente_banca

MODULE main
VAR
-- un cliente dell'oggetto "cliente_banca"
c: cliente_banca;
ASSIGN
-- assegna lo stato iniziale a c
init(c.stato) := nuovo;
-- TRANS
-- descrizione di un "generatore di eventi" (in questo caso `e superflua)
-- case
-- 1: next(c.evento) = paga | next(c.evento) = ritardo |
-- next(c.evento) = null;
-- esac
-- fine MODULE main

LTLSPEC
-- Alcune specifiche soddisfatte
--SE UN CLIENTE E' AFFIDABILE NELLO STATO PRECEDENTE ERA NUOVO O AFFIDABILE
G(c.stato=affidabile -> Y (c.stato=nuovo | c.stato = affidabile))
-- X c.stato=affidabile -> c.stato=nuovo | c.stato = affidabile
--SE NON SI SONO MAI VERIFICATI EVENTI IL CLIENTE E' NUOVO
--G(H(c.evento=null) -> c.stato=nuovo)
--G(c.evento=null) -> c.stato=nuovo
--DOPO IL PRIMO EVENTO IL CLIENTE NON SARA' MAI PIU' NUOVO
--G( c.evento=null | (c.stato!=nuovo S c.evento!=null))
-- (c.evento = null U X G c.stato != nuovo) | G c.evento = null
```

Figura 9: Codice NuSMV del caso del cliente dell'istituto bancario con specifiche LTLB

❖ 6. Conclusioni

Abbiamo intrapreso in questo lavoro un interessante viaggio tra diverse logiche temporali, CTL, CTL*, TRIO e LTLB. Per ognuna di esse, abbiamo analizzato le caratteristiche, i principali pregi e difetti, in particolare confrontandole con la già conosciuta LTL, e le principali possibilità di applicazioni pratiche nell'ambito del Model Checking.

Abbiamo notato come l'utilizzo di CTL sia molto vantaggioso nella specifica di proprietà per documenti di analisi, riuscendo a riempire dei vuoti lasciati da LTL. Sembra in effetti molto consigliabile l'uso parallelo di entrambe le logiche, mentre CTL* risulta troppo generica e poco efficiente.

TRIO appare forse come la logica più interessante, ma anche molto lontana da un'applicazione pratica, nella sua veste formale originaria. Sono pertanto necessarie notevoli semplificazioni, che necessariamente indeboliscono la potenza espressiva del linguaggio. Nonostante ciò, il processo descritto come TRIO2ProMeLa apre importanti prospettive pratiche e merita assolutamente ulteriori analisi, più approfondite.

Infine la descrizione di LTLB ci è sembrata doverosa alla luce delle analogie fra questa e la versione più verosimilmente realizzabile di TRIO. Nonostante LTLB non aggiunga nulla come potere espressivo rispetto a LTL, è innegabile che faciliti la formulazione di specifiche, e proprio per questo motivo negli ultimi tempi molte ricerche si stanno spingendo in profondità nello studio degli operatori del passato.