UNIVERSITÀ DEGLI STUDI DI FIRENZE

Laurea Magistrale in Ingegneria Informatica
Corso di Metodi Formali per la Verifica di Sistemi

Model checking locale per aCTL

A. Rizzo, M. Bruni

Anno accademico 2012/2013

Contents

1 Introduzione

In questa relazione gli autori descriveranno l'algoritmo di model-checking locale per aCTL. Dopo una breve introduzione dove verrano descritti i motivi della verifica formale del modello di un sistema, verranno descritti gli argomenti principali che serviranno per la descrizione dell'algoritmo di model-checking. Parleremo dei Label Transition System (LTS), in italiano sistema di transizione etichettato, utilizzati per la rappresentazione e la modellazione dei sistemi. Verrà definita la logica aCTL che permette di esprimere formalmente le proprietà che un sistema deve rispettare. Infine verrà descritto l'algoritmo di model-checking locale.

1.1 Importanza della correttezza del software

Un programma si dice *corretto* se il suo comportamento è conforme alle specifiche dei requisiti. In altre parole, possiamo dire che un software è corretto se fa esattamente quello per cui è stato creato. L'importanza della correttezza del software diventa un concetto fondamentale soprattutto in ambiente considerato critico. Ad esempio: sistemi di controllo, impianti industriali, sistemi di trasporto, protocolli di comunicazione, etc. . . .

In figura 1 viene mostrato un approccio per la verifica di sistemi. Questo modello di ciclo di sviluppo del software presenta difetti importanti. La verifica del progetto viene fatta in fase avanzata dello sviluppo del codice. Modificare il codice sorgente nelle ultime fasi dello sviluppo potrebbe portare a numerosi problemi e rendere necessarie anche modifiche importanti. Inoltre, come mostrato in figura 2, si può notare in quali fasi di sviluppo viene introdotto il maggior numero di errori e la loro incidenza sul loro costo di risoluzione. L'idea è quindi quella di trovare gli errori già nelle prime fasi del ciclo di sviluppo.

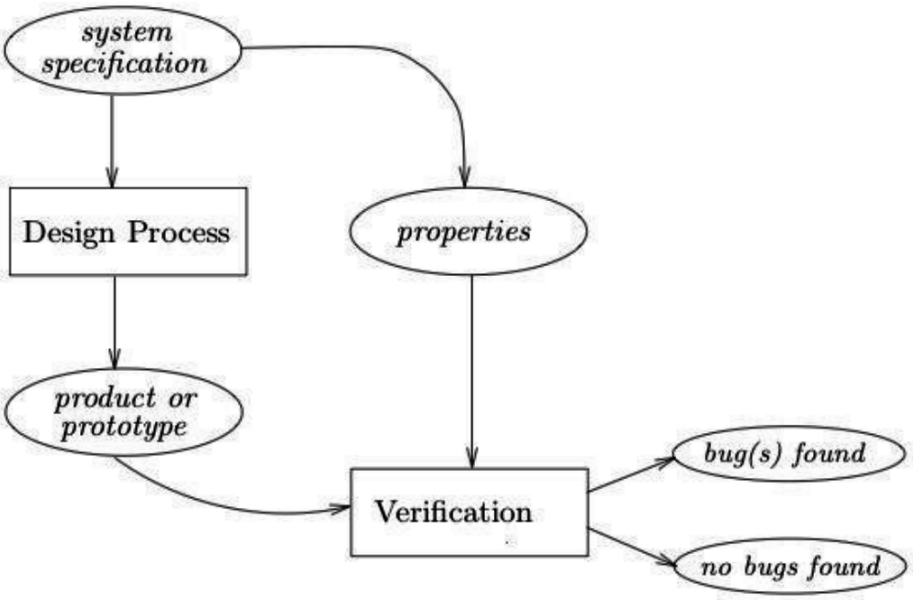

Figure 1: Modello classico di ciclo di sviluppo del software.

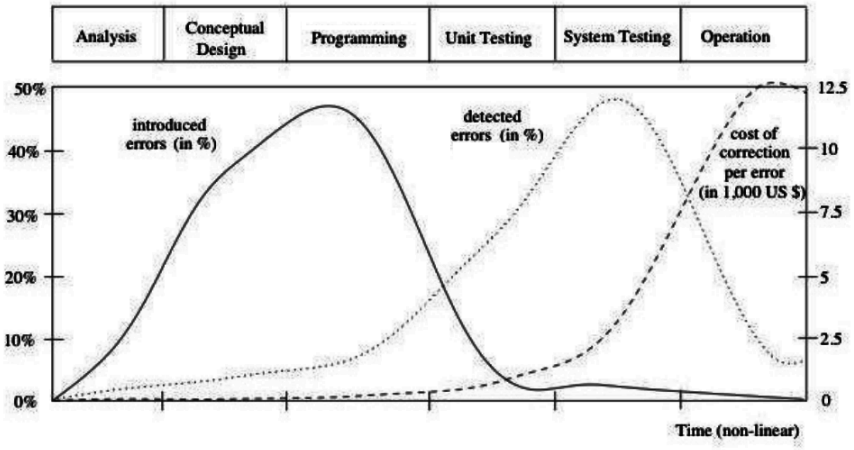

Figure 2: Grafico che mostra: introduzione degli errori, rilevazione degli errori e costo di correzione degli errori durante il ciclo di sviluppo del software.

2 Sistemi di transizione etichettati - (LTS)

La verifica dei sistemi si serve di stumenti automatici. Quindi, viene naturale farsi la seguente domanda:

come possiamo descrivere un sistema software?

Ovviamente non possiamo utilizzare il linguaggio naturale, poichè esso è un linguaggio ambiguo. Abbiamo bisogno di un linguaggio formale che ci permetta di descrivere un sistema software senza ambiguità e che abbia una semantica ben definita. In altre parole abbiamo bisogno di un *modello* del sistema. Per venire incontro a queste necessità sono stati ideati i sistemi di transizione etichettati, in inglese *Label Transition System - (LTS)*. Gli LTS sono una generalizzazione degli *automi a stati finiti - (FSM)* [1], infatti un LTS a differenza di un FSM ha le seguenti proprietà:

- il numero degli stati di un LTS non è necessariamente finito;
- l'insieme delle transizioni di un LTS non è necessariamente finito.

2.1 Definizione formale

Un *sistema di transizione etichettato TS* è una sestupla

$$TS = \langle S, Act, \rightarrow, I, AP, L \rangle \tag{1}$$

dove:

- S è l'*insieme degli stati* del sistema
- Act è l'insieme finito dei simboli che rappresentano le *azioni*. Le azioni possono essere di due tipi: *interne* o *esterne*;
- $\rightarrow \subseteq S \times Act \times S$ è una relazione ternaria detta *relazione di transizione* ed è un sottoinsieme del prodotto cartesiano $S \times Act \times S$.

Esempio: $(p, \lambda, q) \in \rightarrow$ si può scrivere come $p \xrightarrow{\lambda} q$ e significa che il processo p esegue l'azione λ e si comporta come il processo q;

- $I \subseteq S$ è l'*insieme degli stati iniziali*;
- AP è l'*insieme delle proprietà atomiche*. Una proprietà è un predicato P che associa ad ogni elemento x o insieme X un valore di verità;
- L è la *funzione di labeling* o sistema di etichettatura. È definita come

$$L : S \rightarrow 2^{AP} \tag{2}$$

Il dominio della funzione L è l'insieme degli stati del sistema S. Il codominio è l'insieme delle parti delle proprietà atomiche AP. Si ricorda che l'insieme delle parti o insieme potenza di un insieme A è l'insieme

$$2^A = \{X \mid X \subseteq A\} \tag{3}$$

ovvero l'insieme di tutti i sottoinsiemi di A. Per ogni stato $s \in S$, l'insieme $L(s)$ è l'insieme di tutte le proposizione atimiche valide in s.

Esempio

In figura 3 è mostrato il sistema di transizione $TS = \langle S, Act, \rightarrow, I, \ldots \rangle$ di una macchina del caffè. Abbiamo che:

- l'insieme degli stati del sistema: S = { Ready, Select, Coffee, Tea };
- l'insieme delle azioni: Act = { Insert_coin, Select_coffee, Select_tea, Supply_coffee, Supply_tea };
- la relazione di transizione:

  $\rightarrow$ = {(Ready, Insert_coin, Select), (Select, Select_coffee, Coffee), (Select, Select_tea, Tea), (Tea, Supply_tea, Ready), (Coffee, Supply_coffee, Ready)}
- l'insieme degli stati iniziali: I = { Ready }

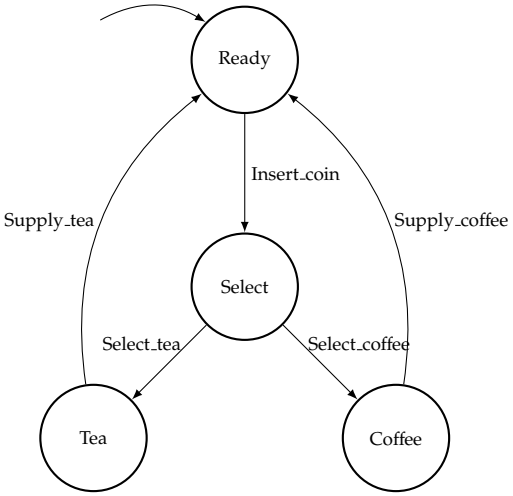

Figure 3: Esempio di uno schema LTS per l'automa di una macchina del caffè.

2.2 Cammino o computazione

In un LTS un *cammino* (o *computazione*) π è una sequenza finita (o infinita) della forma:

$$s_0 \lambda_0 s_1 \lambda_1 s_2 \lambda_2 \ldots\ldots \lambda_{n-1} s_n \tag{4}$$

dove $\forall\, i,\ \exists\, s_i \xrightarrow{\lambda_i} s_{i+1} \in \rightarrow$.

Equivalentemente un cammino può essere scritto nelle seguenti forme:

$$(\, s_0,\ \lambda_0,\ s_1\,), (\, s_1,\ \lambda_1,\ s_2\,),\ \ldots\ldots, (\, s_{n-1},\ \lambda_{n-1},\ s_n\,) \tag{5}$$

oppure

$$s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} s_2 \ \ldots\ldots \xrightarrow{\lambda_{n-1}} s_n \tag{6}$$

Il cammino nullo è la sequenza vuota.

Una cammino si dice *iniziale* se $s_0 \in I$. Una cammino si dice *massimale* se s_n è uno stato terminale, oppure se è un cammino infinito.

Uno stato $s \in S$ è *raggiungibile* in un $TS = \langle\, S,\ Act,\ \rightarrow,\ I,\ AP,\ L\,\rangle$ se esiste

un cammino finito ed iniziale tale che

$$s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} s_2 \ldots\ldots \xrightarrow{\lambda_{n-1}} s_n = s \tag{7}$$

Adesso consideriamo $TS = \langle S, Act, \rightarrow, \ldots \rangle$. Indicheremo con:

- $Post(s, \lambda) = \{ s' \in S \mid s \xrightarrow{\lambda} s' \}$ ovvero l'insieme di tutti gli stati $s' \in S$ raggiungibili dallo stato $s \in S$ facendo un'azione $\lambda \in Act$;
- $Post(s) = \bigcup_{\lambda \in Act} Post(s, \lambda)$ l'insieme di tutti gli stati raggiungibili dallo stato $s \in S$, $\forall \lambda \in Act$;
- $Preset(s, \lambda) = \{ s' \in S \mid s' \xrightarrow{\lambda} s \}$ l'insieme di tutti gli stati che attraverso un'azione $\lambda \in Act$ vanno nello stato $s \in S$;
- $Preset(s) = \bigcup_{\lambda \in Act} Preset(s, \lambda)$ l'insieme di tutti gli stati che evolvono nello stato $s \in S$, $\forall \lambda \in Act$.

Stallo

Un sistema si dice che è in *stallo* (o in deadlock) quando non può eseguire nessuna azione e lo stato in cui si trova non è uno stato terminale.

Osservazioni: un sistema è in stallo se $Post(s) = \emptyset$. Inoltre, se $s \in S$ è uno stato terminale, allora $Post(s) = \emptyset$.

2.3 Determinismo e non determinismo

Un sistema si dice *non deterministico* se può comportarsi in due o più modi differenti con lo stesso input. In formule:

$$\exists\,(s,\lambda)\ : |Post(s,\lambda)| > 1 \tag{8}$$

Un sistema si dice *deterministico* se per ogni stato del sistema, esiste una e una sola azione tale che è possibile determinare in modo univoco lo stato successivo. In formule:

$$\forall s \in S,\ \exists!\,\lambda \in Act\ : |Post(s,\lambda)| = 1 \tag{9}$$

Si possono avere due tipi di determinismo:

- Action Determinism
- AP-Determinism

Un sistema è *action deterministic* (deterministico rispetto alle azioni) allora

$$|I| \leq 1,\ \forall s \in S,\ \forall \lambda \in Act\ \Leftrightarrow |Post(s,\lambda)| \leq 1 \tag{10}$$

ovvero, l'insieme degli stati iniziali deve avere cardinalità al più uno e per ogni coppia stato-azione, l'insieme degli stati successivi deve avere cardinalità al più uno.

Un sistema è *AP-deterministic* (deterministico rispetto alle proprietà atomiche) allora

$$|I| \leq 1,\ \forall s \in S,\ \forall \lambda \in Act\ \Leftrightarrow |Post(s) \cap \{s' \in S \mid L(s') = A\}| \leq 1 \tag{11}$$

ovvero, l'insieme di tutti gli stati successivi allo stato s intersecato con l'insieme degli stati che soddisfano la proprietà atomica A ha cardinalità al più 1.

3 Action-based Computation Tree Logic - (aCTL)

Abbiamo visto che un sistema può essere modellato per mezzo dei sistemi di transizione etichettati (o LTS). Adesso, abbiamo bisogno di un linguaggio formale che ci permetta di scrivere in modo non ambiguo le proprietà del sistema.

3.1 Logiche temporali

Esistono diversi tipi di logiche, tra cui le più note sono la logica modale e la logica temporale. La logica modale permette di esprimere proprietà locali di uno stato. Ad esempio: *"Andrea e Marco sono felici?"*. Il limite di questa logica è che non possiamo sapere se Andrea e Marco continueranno ad essere felici. Questo limite può essere superato attraverso l'utilizzo di logiche temporali.

Le logiche temporali possono essere di due tipi:

- logiche temporali lineari;
- logiche temporali branching.

Nel seguito parleremo di aCTL che è una logica temporale di tipo branching [1].

3.2 Logica aCTL

aCTL sta per Action-based Computation Tree Logic. È una logica temporale di tipo branching, ovvero il tempo è modellato con una struttura ad albero e quindi l'evoluzione futura non è determinata in modo univoco. Questo approccio si avvicina alla vera esecuzione di un programma in cui alcune scelte vengono fatte durante l'esecuzione del processo. Le formule di aCTL si riferiscono esplicitamene alle azioni che il sistema può fare.

3.2.1 Sintassi di aCTL

Distinguiamo due categorie sintattiche:

- *state formule* ϕ;
- *path formule* φ:

Il linguaggio è generato dalla seguente grammatica:

State-formula

$$\phi ::= true \mid \neg\phi \mid AP \mid \phi_1 \ \wedge \phi_2 \mid \exists\varphi \mid \forall\varphi \tag{12}$$

- *true* : è l'assioma *vero*;
- AP : sono le proposizioni atomiche. Ad esempio: la state formula $\phi = a$ è vera per lo stato s se $a \in L(s) \subseteq AP$;
- $\neg\phi$: è la negazione di ϕ;
- $\phi_1 \wedge \phi_2$: è la congiunzione di ϕ_1 e ϕ_2 ;
- $\exists\varphi$: è il quantificatore esistenziale. Il suo significato è che esiste almeno un cammino (o computazione) che partendo dallo stato attuale permette di raggiungere uno stato che soddisfa φ;
- $\forall\varphi$: è il quantificatore universale. Il suo significato è che tutti i cammini che partono dallo stato corrente soddisfano la proprietà φ.

Path-formula

$$\varphi ::= X_A\phi \mid \phi \ {}_A\mathcal{U} \ \psi \mid \phi \ {}_A\mathcal{U}_B \ \psi \tag{13}$$

- $X_A\phi$: è l'operatore *next* associato all'azione A. La formula è soddisfatta se a partire dallo stato corrente è possibile fare un'azione A ed andare in uno stato che soddisfa la proprietà ϕ;

- $\phi\ {}_A\mathcal{U}\ \psi$: è l'operatore *until* associato all'azione A. La formula è soddisfatta se si percorre un cammino (anche nullo) eseguendo azioni in A e tutti gli stati soddisfano la proprietà ϕ fino a quando non si arriva in uno stato che soddisfa la proprietà ψ;

- $\phi\ {}_A\mathcal{U}_B\ \psi$: è loperatore *until* associato all'azione A e B. La formula è soddisfatta se si percorre un cammino (anche nullo) eseguendo azioni in A e tutti gli stati soddisfano la proprietà ϕ fino a quando non si arriva in uno stato che soddisfa la proprietà ψ attraverso l'esecuzione di una azione in B.

Le formule aCTL sono interpretate sui sistemi di transizione etichettati. In particolare è bene fare una distinzione tra le state formule e path formule. Le state formule sono valutate sugli stati del LTS, viceversa le path formule sono valutate sui cammini o computazioni del LTS. Come si può notare, una path formula può occorrere in una state formula come parametro degli operatori $\exists$ e $\forall$.

Esistono altri operatori, detti operatori derivati, che permettono di riscrivere le formule in altre formule equivalenti che possono risultare più comprensibili.

$$false \equiv \neg true \tag{14}$$

$$\phi_1 \wedge \phi_2 \equiv \neg\,(\neg\phi_1 \vee \neg\phi_2) \tag{15}$$

$$\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2 \tag{16}$$

$$\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge\ (\phi_2 \Rightarrow \phi_1) \tag{17}$$

3.2.2 Operatori

Diamo ora una classificazione degli operatori. Gli operatori possono essere di due tipi: logici e temporali.

Operatori logici

Gli operatori logici sono i seguenti:

$$\neg,\ \wedge,\ \vee,\ \Rightarrow,\ \Leftrightarrow \tag{18}$$

Operatori temporali

Gli operatori temporali sono

$$\forall,\ \exists,\ X,\ U,\ \Diamond,\ \Box \tag{19}$$

in particolare gli operatori $\exists$ e $\forall$ sono dei quantificatori di cammino.

3.2.3 Semantica

Come anticipato le formule aCTL sono interpretate sui sistemi di transizione etichettati. Un sistema di transizioni, come abbiamo visto, è una sestupla $TS = \langle\ S, Act, \rightarrow, I, AP, L\ \rangle$ dove S è l'insieme degli stati, Act è l'insieme dei simboli che rappresentano le azioni, $\rightarrow$ è l'insieme delle relazioni di transizione, I è l'insieme degli stati iniziali, AP è l'insieme delle proprietà atomiche e L è la funzione di labeling.

Sia quindi TS tale sistema di transizioni con s $\in$ S e $\Phi \in F$ dove F è l'insieme delle formule aCTL. La semantica di una formula temporale è definita dalla soddisfazione della seguente relazione:

$$\models: (TS \times S \times F) \rightarrow \{true, false\} \tag{20}$$

Una proposizione atomica è vera (*true*) in uno stato s_i quando:

$$TS, s_i \models p \text{ se e solo se } p \in L(s_i) \tag{21}$$

La relazione di implicazione semantica può essere definita per induzione strutturale su Φ.

$$TS, s_i \models true \qquad \forall s \in S \tag{22}$$

$$TS, s_i \models \neg\Phi \quad \Leftrightarrow \quad \neg TS, s_i \models \Phi \tag{23}$$

$$TS, s_i \models \Phi \wedge \Psi \quad \Leftrightarrow \quad TS, s_i \models \Phi \text{ and } TS, s_i \models \Psi \tag{24}$$

$$TS, s_i \models \Phi \vee \Psi \quad \Leftrightarrow \quad TS, s_i \models \Phi \text{ or } TS, s_i \models \Psi \tag{25}$$

$$TS, s_i \models \Phi \Rightarrow \Psi \quad \Leftrightarrow \quad (\neg TS, s_i \models \Phi \text{ or } TS, s_i \models \Psi) \tag{26}$$

$$TS, s_i \models \Phi \Leftrightarrow \Psi \quad \Leftrightarrow \quad \begin{array}{l}(TS, s_i \models \Phi \text{ or } TS, s_i \models \Psi) \\ \vee\ (\neg TS, s_i \models \Phi \text{ or } \neg TS, s_i \models \Psi)\end{array} \tag{27}$$

Gli operatori temporali, considerando $\pi = (s_0, s_1, \ldots, s_n) \in Path(s)$ un generico cammino avente origine dallo stato $s_i \in S$, hanno la seguente semantica:

$$TS, s_i \models \forall\, \varphi \quad \Leftrightarrow \quad \forall \pi \in Path(s_i) : \pi \models \varphi \tag{28}$$

$$TS, s_i \models \exists\, \varphi \quad \Leftrightarrow \quad \exists \pi \in Path(s_i) : \pi \models \varphi \tag{29}$$

Rimane infine da definire la semantica per gli operatori di χ e di $\mathcal{U}$ lungo un cammino π:

$$\pi = s_0\alpha_0 s_1 \ldots \models \chi_A\ \Phi \quad \Leftrightarrow \quad \alpha_0 \in A \text{ and } TS, s_1 \models \Phi \tag{30}$$

$$\pi = s_0\alpha_0 s_1 \ldots \alpha_{n-1}s_n \models \Phi_A\ \mathcal{U}\ \Psi \quad \Leftrightarrow \quad \begin{array}{l} \exists j \geq 0 :\ \pi(j) \models \Psi \text{ and} \\ \forall 0 \leq i < j :\ \pi(i) \models \Phi \text{ and } \alpha_i \in A) \end{array} \tag{31}$$

$$\pi = s_0\alpha_0 s_1 \ldots \alpha_{n-1}s_n \models \Phi_A\ \mathcal{U}_B\ \Psi \quad \Leftrightarrow \quad \begin{array}{l} \exists j \geq 1 :\ \pi(j) \models \Psi \text{ and } \alpha_{j-1} \in B \\ \forall 0 \leq i < j :\ \pi(i) \models \Phi \text{ and} \\ \forall 0 \leq i < j - 1\ \alpha_i \in A \end{array} \tag{32}$$

In base a quanto è stato appena enunciato è facile verificare anche le formule

$$TS, s_i \models \forall\ \Box\ \Phi \tag{33}$$

$$TS, s_i \models \exists\ \Box\ \Phi \tag{34}$$

$$TS, s_i \models \forall\ \Diamond\ \Phi \tag{35}$$

$$TS, s_i \models \exists\ \Diamond\ \Phi \tag{36}$$

Queste formule sono infatti sono facilmente ricavabili nel modo seguente:

$$\exists\ \Diamond_A \phi \ \equiv\ \exists\ (true\ {}_A\mathcal{U}\ \phi) \tag{37}$$

$$\forall\ \Diamond_A \phi \ \equiv\ \forall\ (true\ {}_A\mathcal{U}\ \phi) \tag{38}$$

$$\exists\ \Box_A \phi \ \equiv\ \neg\ \forall\ \Diamond_A\ \neg\ \phi \tag{39}$$

$$\forall\ \Box_A \phi \ \equiv\ \neg\ \exists\ \Diamond_A\ \neg\ \phi \tag{40}$$

$$\forall[\Phi\ \mathcal{U}\ \Psi] \equiv \neg\exists\Box\neg\Psi \wedge \neg\exists(\neg\Psi\mathcal{U}(\neg\Phi \wedge \neg\Psi)) \tag{41}$$

4 Model checking

Il *model checking* è un metodo per la verifica dei sistemi. Fu introdotto da E. Clarke, A. Emerson e J. Sifakis con cui si aggiudicarono nel 2007 il A.M. Turing Award. Il metodo del model checking è schematizzato in figura 4. Un *model checker* riceve in *input* il modello del sistema software ed i requisiti che deve soddisfare. In *ouput* restituirà un valore booleano: *true* o *false* nel caso in cui la proprietà che si è andata a verificare sia rispettivamente soddisfatta o non soddisfatta. Se la proprietà non è soddisfatta viene fornito un *controesempio*.

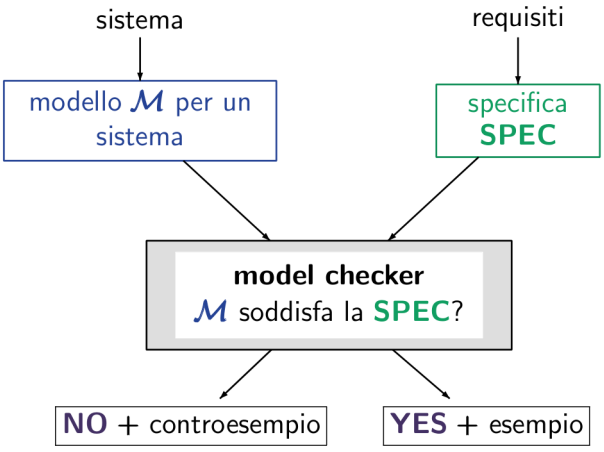

Figure 4: Model checking.

Formalmente, la verifica di una proprietà tramite model checking consiste: dato un modello del sistema M con stato iniziale s e data una proprietà ϕ che il sistema deve soddisfare, fare il model checking consiste nel dire se:

$$M, s \models \phi \tag{42}$$

$$M, s \not\models \phi \tag{43}$$

Il modello del sistema su cui vogliamo fare model checking è descritto

mediante un sistema di transizione etichettato (LTS). Le proprietà da verificare sono descritte attraverso un linguaggio formale in logica temporale, nel nostro caso aCTL.

4.1 Model checking globale e locale

Come abbiamo detto i sistemi sono una composizione di più modelli che descrivono il comportamento di un processo. Un limite del model checking è il cosidetto fenomeno dell'*esplosione dello spazio degli stati*. La rappresentazione di un sistema con un numero elevato di stati (ad esempio dell'ordine di 10^{100}) richiede molta memoria.

Possiamo distinguere due approcci di model checking:

- model checking globale
- model checking locale

Il *model checking globale*, consiste nell'andare a verificare se tutti gli stati del sistema preso in considerazione, verificano una proprietà. Questa è un'operazione computazionalmente onerosa in quanto verrà generato l'intero spazio degli stati. Inoltre, vengono generati anche gli stati che non sono rilevanti al fine di dimostrare la soddisfacibilità della formula. L'approccio utilizzato è quello di una ricerca *bottom-up* a partire dalle foglie dell'albero fino a risalire alla radice.

Viceversa, il *model checking locale* genera solo gli stati che servono effettivamente a verificare la proprietà. In questo caso viene adottato un approccio *top-down*, e cioè a partire da un nodo radice si percorre l'albero verso le foglie.

4.1.1 Existential normal form (ENF)

Al fine di sviluppare un model checker, viene utilizzata una notazione alternativa alle formule aCTL, l'Existential Normal Form (ENF), che permette una più semplice trasposizione in forma algoritmica. Qualsiasi formula aCTL può essere infatti scritta come combinazione delle formule ENF.

$$\Phi ::= true \mid ap \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\, \chi_A\, \Phi \mid \exists(\Phi_{1_A}\, \mathcal{U}\, \Phi_2) \mid \exists(\Phi_{1_A}\, \mathcal{U}_B\, \Phi_2) \mid \exists\Box_A\Phi \quad (44)$$

I passi da eseguire per verificare se uno stato, appartenente a TS, verifica una formula aCTL $\hat{\Phi}$, sono i seguenti:

1. si converte la formula $\hat{\Phi}$ nella sua equivalente Φ in ENF
2. si calcola ricorsivamente l'insieme $Sat(\Phi) = \{s \in S | s \models \Phi\}$
3. TS, $s_i \models \Phi$ se e solo se $s_i \in Sat(\Phi)$

Definiamo quindi per casi, i possibili valori assumibili da Sat(Φ):

$$Sat(true) \rightarrow S \tag{45}$$

$$Sat(ap) \rightarrow \{s | ap \in L(s)\} \tag{46}$$

$$Sat(\Phi \wedge \Psi) \rightarrow Sat(\Phi) \cap Sat(\Psi) \tag{47}$$

$$Sat(\neg\Phi) \rightarrow S\ Sat(\Phi) \tag{48}$$

$$Sat(\exists \mathcal{X}_A \Phi) \rightarrow \{s | Post_A(s) \cap Sat(\Phi) \neq \emptyset \tag{49}$$

$Sat(\exists \Phi_A \mathcal{U} \Psi) \rightarrow$ il più piccolo sottoinsieme T ⊆ S,tale che: (50)

1) $Sat(\Psi) \subseteq T$

2) $se\ s \in Sat(\Phi) \wedge Post_A(s) \cap T \neq \emptyset \Rightarrow s \in T$

$Sat(\exists \Phi_A \mathcal{U}_B \Psi) \rightarrow$ il più piccolo sottoinsieme T ⊆ S,tale che: (51)

1) $se\ s \in Sat(\Phi) \wedge Post_B(s) \cap Sat(\Psi) \neq \emptyset \Rightarrow s \in T$

2) $se\ s \in Sat(\Phi) \wedge Post_A(s) \cap T \neq \emptyset \Rightarrow s \in T$

$Sat(\exists \Box_A \Phi) \rightarrow$ il più piccolo sottoinsieme T ⊆ S,tale che: (52)

1) $Sat(\Psi) \subseteq T$

2) $se\ s \in T \Rightarrow Post_A(s) \cap T \neq \emptyset$

5 Algoritmi

L'obiettivo del model checking locale per aCTL è il seguente: data una struttura $\mathcal{T}$ e una formula aCTL f, determinare se $\mathcal{T}$ soddisfa f. Verrà adesso introdotta l'implementazione di un algoritmo per la risoluzione del problema.

L'algoritmo dovrà soddisfare le seguenti specifiche:

- input: un LTS $\mathcal{T}$, uno stato s, una formula CTL f;
- output: true $\Leftrightarrow$ $\mathcal{T}, s \models f$.

L'algoritmo dovrà inoltre operare secondo una modalità *goal-oriented*, e cioè con approccio top-down, per decidere se $\mathcal{T}$, $s \models f$. Gli stati che dovranno essere controllati durante la sua esecuzione dovranno essere quindi solo quelli strettamente necessari a raggiungere lo stato di goal.

Per poter eseguire l'algoritmo nel minor tempo possibile si è reso necessario l'utilizzo di una variabile globale per tenere traccia di tutte le informazioni raccolte durante l'esecuzione. Questa struttura è stata definita nel modo seguente:

$$info : S \; x \; sub(f) \rightarrow \{0, 1, \bullet\}, \tag{53}$$

dove $sub(f)$ rappresenta l'insieme delle sottoformule di f. I valori assumibili dalla variabile rappresentano la soddisfacibilità di una determinata formula. Se uno stato $s \in$ S non soddisfa la formula f, si avrà di conseguenza che info(s, f) = 0. Nel caso in cui s soddisfi la formula f si avrà che info(s, f) = 1, mentre nel caso in cui non sia ancora stato possibile determinarlo si avrà info(s, f) = $\bullet$.

Formalmente il significato della variabile info è dato dalla seguente invariante che dovrà essere sempre valida durante l'esecuzione dell'algoritmo:

$$I \stackrel{def}{=} \forall s, f : \; (info(s, f) = 0 \Rightarrow \mathcal{T}, s \not\models f) \wedge (info(s, f) = 1 \Rightarrow \mathcal{T}, s \models f) \tag{54}$$

Viene mostrato in seguito il codice relativo allo schema generale dell'algoritmo:

```
1  Check(s, Φ) {
2      if info(s, Φ) ≠ •
3          return;
4      switch(Φ) {
5          case a:
6              if s ∈ 𝕃(p) then
7                  info(s, Φ) := 1;
8              else
9                  info(s, Φ) := 0;
10         case ¬ Φ′:
11             Check(s, Φ′);
12             info(s, ¬ Φ′) = ¬ info(s, Φ′);
13         case Φ1 ∧ Φ2:
14             Check(s, Φ1);
15             if info(s, Φ1) = 0 then
16                 info(s, Φ1 ∧ Φ2) = 0;
17             else {
18                 Check(s, Φ2);
19                 info(s, Φ1 ∧ Φ2) = info(s, Φ2);
20             }
21         case ∃χA Φ′:
22             foreach s′ ∈ {s′|∃α ∈ A : s --α--> s′}{
23                 Check(s′,Φ′);
24                 if (info(s′,Φ′)==1) {
25                     info(s′,∃χA Φ′) = 1;
26                     return;
27                 }
28             }
29             info(s, ∃χA Φ′) = 0;
30         case ∀χA Φ′:
31             init(s) = {α|∃s′ : s --α--> s′}
32             if ( init(s) ⊈ A ) then {
33                 info(s, ∀χA Φ′) = 0;
34                 return;
35             }
36             foreach s′ ∈ {s′|∃α ∈ A : s --α--> s′}{
37                 Check(s′,Φ′);
38                 if (info(s′,Φ′)==0) then {
39                     info(s′,∃χA Φ′) = 0;
40                     return;
41                 }
42             }
43             info(s, ∃χA Φ′) = 1;
44         case ∃Φ1A 𝒰Φ2:
45             CheckEU(s, A, Φ1,Φ2);
46         case ∀Φ1A 𝒰Φ2:
47             CheckAU(s, A, Φ1,Φ2);
48     }
49 }
```

Code 1: Schema generale dell'algoritmo

L'implementazione in Code 1, affronta in modo ricorsivo tutte le formule previste dalla sintassi da aCTL. Nel caso delle proprietà $\exists\Phi_{1A}\mathcal{U}\Phi_2$ e $\forall\Phi_{1A}\mathcal{U}\Phi_2$,

l'implementazione è stata suddivisa nelle funzioni dedicate CheckEU e CheckAU.

```
1  CheckEU(s, A, Φ1, Φ2) {
2      V = ∅;                              // stati visitati
3      E = {s};                            // stati in corso di visita
4      goal = •;
5      while (goal == • ∧ E ≠ ∅) {
6          s'= pick(E);
7          E = E \ {s'};
8          V = V ∪ {s'};
9          Check(s', Φ2);
10         if (info(s', Φ2) == 1) then {
11             goal = s';
12             break;
13         }
14         Check(s', Φ1);
15         if (info(s', Φ1) == 1) then {
16             E = E ∪{s'' | ∃α ∈ A : s' --α--> s''} \ V;
17         }
18     }
19     if (goal ≠ •) then
20         info(s, ∃Φ1 A U Φ2) = 1;
21     else
22         info(s, ∃Φ1 A U Φ2) = 0;
23 }
```

Code 2: Implementazione di CheckEU

Il codice precedente esegue una visita (parziale) in profondità dalla radice s. La ricerca viene interrotta appena viene trovato uno stato s che soddisfa $\exists\Phi_1 A \mathcal{U} \Phi_2$, e cioè quando info($s$, $\exists\Phi_1 A \mathcal{U} \Phi_2$) = 1, oppure quando viene trovato uno stato s che soddisfa Φ_2. Se info(s, $\exists\Phi_1 A \mathcal{U} \Phi_2$) = 0, oppure se sia Φ_1 che Φ_2 non sono soddisfatti da uno stato s, allora non è necessario proseguire. Al fine di implementare un ricerca in profondità, la struttura E mantiene traccia degli stati in corso di visita e viene gestita in modalità LIFO.

Ignorando il tempo richiesto per processare le chiamate alla funzione di Check, possiamo osservare che il tempo di esecuzione di CheckEU è di $O(|\mathcal{T}|)$. Per questo motivo nel caso in cui si abbiano più formule annidate del tipo $\exists[\_\mathcal{U}\_]$ non è possibile effettuare model checking in tempo lineare.

Consideriamo per esempio la formula $\exists f_1 A \mathcal{U} f_2$ dove f_1 e f_2 contengono a loro volta sottoformule del tipo $\exists[\_\mathcal{U}\_]$. Per decidere se $\mathcal{T}, s \models \exists f_1 A \mathcal{U} f_2$, nel caso peggiore potrebbe essere necessario controllare f_1 e f_2 per ogni stato e ognuna di queste ricerche potrebbe richiedere un attraversamento completo

del sistema di transizioni impiegando un tempo quadratico[2].

Per poter migliorare le performance di CheckEU è necessario provvedere a memorizzare nella struttura di info maggiori informazioni sulla propietà in esame. Nell'implementazione precedente infatti la struttura di info veniva aggiornata solamente per lo stato di s. La funzione può essere quindi modificata per salvare informazioni sulla validità della proprietà per tutti gli stati visitati durante l'esecuzione di CheckEU. Una possibile modifica di CheckEU viene mostrata in seguito.

```
1  CheckEU(s, A, Φ1, Φ2) {
2      V = ∅;                              // stati visitati
3      E = {s};                            // stati in corso di visita
4      goal = •;
5      while (goal == • ∧ E ≠ ∅) {
6          s'= pick(E);
7          E = E \ {s'};
8          V = V ∪ {s'};
9          Check(s', Φ2);
10         if (info(s', Φ2) == 1) then {
11             goal = s';
12             break;
13         }
14         Check(s', Φ1);
15         if (info(s', Φ1) == 1) then {
16             E = E ∪{s'' | ∃α ∈ A : s' --α--> s''} \ V;
17         }
18     }
19     if (goal == •) then {
20         foreach s' ∈ V
21             info(s, ∃Φ1 A U Φ2) = 0;
22     } else  {
23         foreach s' in ∈ V {
24             if s ~*~> goal then
25                 info(s, ∃Φ1 A U Φ2) = 1;
26             else
27                 info(s, ∃Φ1 A U Φ2) = 0;
28         }
29     }
30 }
```

Code 3: Implementazione modificata di CheckEU

La parte terminale della funzione è stata modificata per aggiornare tutti i nodi visitati durante l'esecuzione dell'algoritmo. Se al termine di CheckEU la variabile di goal è ancora uguale a •, significa che nessun cammino soddisfa la proprietà in esame. Se invece la variabile di goal è stata modificata, significa che

conterrà al suo interno lo stato terminale del cammino che soddisfa la proprietà. Indicando con s $\overset{*}{\rightsquigarrow}$ goal il cammino dallo stato di partenza a quello di goal, si aggiorna lungo il cammino la variabile info(s, $\exists\Phi_1 A\mathcal{U}\Phi_2$) = 1, ponendola pari a 0 per tutti gli stati che non ne fanno parte. Il cammino può essere ricavato risalendo all'indietro dallo stato di goal qualora vengano memorizzate anche le transizioni inverse. Un metodo alternativo alla memorizzazione delle transizioni inverse è quello proposto da Vegauwen e Lewi [2] che incorpora il calcolo del cammino all'interno della ricerca depth-first.

L'implementazione della funzione di CheckAU è molto più semplice di quella di CheckEU. Infatti nel caso in cui, durante una ricerca depth-first, venga trovato un cammino ciclico che soddisfa Φ_1 ma non Φ_2 possiamo concludere che la formula $\forall\Phi_1 A\mathcal{U}\Phi_2$ è falsa per tutti gli stati analizzati.

```
1  CheckAU(s, A, Φ1, Φ2) {
2      V = ∅;                              // stati visitati
3      E = {s};                            // stati in corso di visita
4      goal = •;
5      while (goal == • ∧ E ≠ ∅) {
6          s'= pick(E);
7          E = E \ {s'};
8          V = V ∪ {s'};
9          Check(s', Φ2);
10         // se info = 1, il ramo e' verificato
11         if (info(s', Φ2) == 0) then {
12             Check(s', Φ1);
13             if (info(s', Φ1) == 0) then {
14                 goal = 0;
15                 break;                          // esce dal ciclo
16             } else {
17                 int(s') = {α | ∃s' --α--> s''};
18                 if (int(s') ⊈ A) {
19                     goal = 0;
20                     break;                      // esce dal ciclo
21                 }
22                 E = E ∪{s'' | ∃α∈A : s' --α--> s''} \ V;
23             }
24         }
25     }
26     if (goal == 0) then
27         info(s, ∀Φ1 A𝒰Φ2) = 0;
28     else
29         info(s, ∀Φ1 A𝒰Φ2) = 1;
30 }
```

Code 4: Implementazione di CheckAU

Nel caso in cui si voglia salvare le informazioni ricavate su tutti i nodi visitati è sufficiente modificare la funzione nel modo seguente.

```
1  CheckAU(s, A, Φ1, Φ2) {
2      V = ∅;                              // stati visitati
3      E = {s};                            // stati in corso di visita
4      goal = •;
5      while (goal == • ∧ E ≠ ∅) {
6          s'= pick(E);
7          E = E \ {s'};
8          V = V ∪ {s'};
9          Check(s', Φ2);
10         // se info = 1, il ramo e' verificato
11         if (info(s', Φ2) == 0) then {
12             Check(s', Φ1);
13             if (info(s', Φ1) == 0) then {
14                 goal = 0;
15                 break;                          // esce dal ciclo
16             } else {
17                 int(s') = {α | ∃s' --α--> s''};
18                 if (int(s') ⊈ A) {
19                     goal = 0;
20                     break;                      // esce dal ciclo
21                 }
22                 E = E ∪{s'' | ∃α ∈ A : s' --α--> s''} \ V;
23             }
24         }
25     }
26     if (goal == 0) then
27         foreach s'' ∈ V:
28             info(s'', ∀Φ1 A U Φ2) = 0;
29     else
30         foreach s'' ∈ V:
31             info(s'', ∀Φ1 A U Φ2) = 1;
32 }
```

References

[1] R. D. Nicola and R. Pugliese, *Algebre di processo come modelli per sistemi interattivi*. Università degli Studi di Firenze, 2012.

[2] B. Vergauwen and J. Lewi, "A linear local model checking algorithm for ctl," vol. 715, pp. 447–461, 1993. [Online]. Available: http://dx.doi.org/10.1007/3-540-57208-2_31