

A.A. 2021/22

FONDAMENTI DI  
ANALISI E VERIFICA DEL SOFTWARE

PROF. ISABELLA MASTROENI

FABS :)

## NOTA

**Questi appunti/sbobinatura/versione “discorsiva” delle slides sono per mia utilità personale,**  
quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che  
anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 

Appunti e puttanate sono scritte *principalmente* in questo stile.

**Questo file fa parte della mia collezione di sbobinature,  
che è disponibile (e modificabile!) insieme ad altre in questa repo:  
<https://github.com/fabfabretti/sbobinamento-seriale-uniVR>**

## INDICE

|                       |    |
|-----------------------|----|
| NOTA .....            | 2  |
| Indice.....           | 3  |
| Analisi dinamica..... | 5  |
| Testing.....          | 6  |
| Monitoring.....       | 14 |
| Model checking.....   | 19 |
| Model checking.....   | 25 |



## ANALISI DINAMICA

### 22 Analisi dinamica 1 Testing

Sarebbero le basi fondamentali teoriche del testing. Vediamo due tipi di analisi di questo tipo:

#### *Testing*

Analisi che guarda l'esecuzione intera del programma; come l'analisi statica, cerca di dare risposte sull'intera esecuzione e si fanno scelte di approssimazione.

Se nell'analisi statica la scelta sta nell'accettare risposte approssimate (es. prendo un intervallo riempiendo i buchi).

Il testing invece fa una scelta diversa: si decide di guardare esattamente l'esecuzione del programma, ma guardando un insieme intero (e quindi decidibile)

#### *Monitoring*

E' un controllore che mantiene controllato se un'esecuzione soddisfa determinati limiti o viola determinate proprietà.

Riguarda programmi in esecuzione; di solito si tratta di software che girano insieme al programma che stiamo analizzando. Un esempio sono le tecniche di intrusion detection.

Ce ne sono altre che non vediamo in quanto sono metodi molto più pratici e collegati a strumenti concreti:

- Debugging
- Emulation/virtualization
- Profiling/tracing
- **Slicing**

Ho una variabile di interesse da osservare in output; il codice è molto grande quindi fare un'analisi statica sarebbe troppo costoso... allora faccio lo slicing per estrarre dal programma originale solo quella sottoporzione di programma che ha effetto sulla variabile che mi interessa, e elimino tutto il resto.

- **Disassembly e decompilation**

Servono in quei contesti in cui devo analizzare codice binario; dato che tutti gli altri strumenti che ho sono per codice più ad altro livello, cerco di ricreare il codice ad alto livello.

# TESTING

## Def Testing

Consiste nell'eseguire il programma su un certo sottoinsieme di valori di input.

La definizione standardizzata dall'IEEE è: "Il processo di analisi di un software per rilevare le differenze fra le condizioni esistenti e richieste (ovvero difetti, errori e bug) e per valutare le future del software".

E' una tecnica **dinamica**, perché **esegue il programma**. Questo significa che **passa per la non decidibilità**, nel senso che posso togliere un pezzo di problema testando su un insieme limitato di input, ma comunque se il problema non termina sono fottuto.

Hanno due problemi grossi:

- Devono capire se **quando una computazione ci mette tanto** è perché il programma non termina o è semplicemente lento.
- Usiamo un **subset molto piccolo dei possibili input**: quindi, dobbiamo supporre che il comportamento di ogni input possibile sia coerente con il comportamento osservato.

## Obiettivi del testing

Abbiamo due tipi di testing con diversi obiettivi:

- **Debug testing**: serve a trovare difetti e errori.
- **Acceptance testing**: serve a fornire una validazione dell'affidabilità del software, secondo diversi termini:
  1. Correttezza
  2. Sicurezza
  3. politiche di accesso
  4. ...qualunque altre proprietà definibile sul programma.

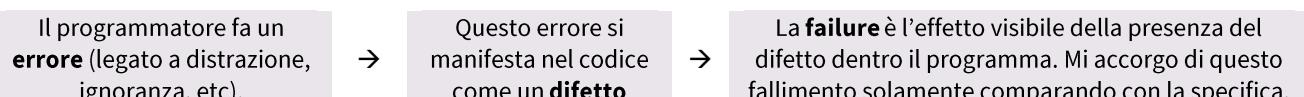
In entrambi i casi, il testing ha l'obiettivo generale di validare delle specifiche. Dovvrò trovare un insieme di input grande abbastanza da trovare questi errori che fanno fallire.

## Terminologia

In italiano, purtroppo, non abbiamo lo stesso livello di distinzione che troviamo in inglese.

- **Mistake**: azione umana che ha prodotto un errore; per esempio, l'errore di programmazione che fa sì che ho scritto in modo sbagliato una variabile, etc.  
Il mistake è la causa dell'errore/fault.
- **Fault (o defect)**: è l'errore vero e proprio presente del codice (magari frutto di azione umana).
- **Failure**: se il programma da qualche parte ha un errore, comunque non è detto che il problema diventi evidente; la failure è l'incapacità del sistema o di un componente di funzionare dentro i propri requisiti.  
Un problema non testato adeguatamente potrebbe contenere dei difetti (fault) ma non dare fallimenti.
- **Errore**: qui si intende l'errore come misura; è la differenza fra una condizione o un valore computato, osservato o misurato rispetto al valore o alla condizione vera, specificata o teoricamente corretta. E' la differenza fra quello che misuriamo e quello che ci aspettiamo.
- **Specifiche**: è il documento che in modo più o meno formale mi permette di descrivere quali sono le caratteristiche che vorremmo fossero dimostrate e verificate nel nostro programma.

Quindi:



Se il testing non arriva a rendere evidente il difetto.. questo rimane latente, e può evidenziarsi nel momento meno opportuno. Quindi, il testing ha l'obiettivo di rilevare i fallimenti.

E' comunque una cosa diversa dal debugging! Il testing rivela esclusivamente il fallimenti; **non c'è ancora il passaggio a "capire dove è avvenuto l'errore"**. Il testing si limita a rilevare l'esistenza del difetto; solo con il debugging potrò poi cercare dove si trova.

### Esempio

- **Mistake:** errore del programmatore che ha scritto il programma che vuole duplicare un valore come “moltiplicazione del valore per se stesso”.
- **Difetto:** c'è scritto “`**` anziché “`+`”
- **Failure:** usando in input dei valori la cui potenza di due e il doppio sono diversi, noterò che l'output è diverso rispetto a quello che mi aspettavo.
  1. Per esempio, 0 o 2 non evidenziano fallimento
  2. 1 invece sì (ottengo 1 anziché 2)

```
program double (in,out)
var x, y: integer;
begin
  read(x);
  y := xx;
  write(y);
end
```

## Decidibilità

Quali sono gli aspetti decidibili e non decidibili nel testing?

Per formalizzare questi teoremi devo prima formalizzare il concetto di testing.

### Formalizzazione del concetto di testing

#### Def Testing (formale)

Presa una funzione  $P: D^{input} \rightarrow R^{output}$ , descriviamo il programma come  $P(d)$ , ovvero la funzione che calcola sull'input  $d$ .

#### Def Correttezza su un input

Per definire la correttezza, diciamo che  $ok(P, d)$  è true quando  $P(d)$  restituisce esattamente il risultato atteso rispetto a qualche specifica.

#### Def Correttezza del programma (generale)

Quindi, il termine è corretto in termini assoluti (aka indipendentemente dall'input) quando per ogni possibile valore in input, il risultato che ottengo applicando  $P$  a quell'input è esattamente quello che mi aspetto.

$$ok(P) \text{ iff } \forall d \in D . ok(P, d)$$

Ovviamente, se  $D$  è infinito questo problema non è decidibile.

#### Def Esecuzione di un test

Supponiamo di rappresentare un test come  $T$ . Il test  $T$  è un sottoinsieme dei dati  $D$ , quindi  $T \subseteq D$ . I singoli dati di input di test saranno  $t \in T$ . Eseguire il test significa eseguire  $P(t)$  su tutti i  $t \in T$ .

#### Correttezza su un test

$ok(P, T)$  indica che il test è corretto su tutti i test per tutti gli elementi  $t \in T$ .

#### Test di successo

L'obiettivo del testing è **rilevare i fallimenti**: un test è di successo quando trova dei fallimenti.

Un test  $t$  ha successo per  $P$ , ovvero  $success(T, P)$ , se rileva uno o più fallimenti in  $P$ , ovvero se il programma non è sound su  $P$ :  $\exists t \in T . \neg ok(P, t)$

Se, al contrario, il programma si comporta sempre bene – cioè  $\forall t \in T . ok(P, t)$ , allora non posso dire niente.

#### Test ideale

Chiamiamo test ideale il test per il quale se il test è  $ok$  per tutti i valori, allora il programma è  $ok$

$$ok(P, T) \Rightarrow ok(P)$$

In questo modo, se fallisce un test ideale allora sappiamo che il programma è sicuramente corretto.

#### Criterio di selezione di test

Chiaramente non possiamo elencare tutti i possibili valori su cui facciamo i test, quindi ci servono dei criteri in base ai quali selezionare gli input.

### Def Criterio di test

Un criterio di test  $C$  per  $P$  è un insieme di predicati sul dominio  $D$  del programma. Seleziona dati di test usando i criteri in  $C$ .

Un insieme di test è selezionato da un criterio  $C$ , ovvero  $selected(C, T)$ , se per ogni elemento  $t$  esiste un criterio soddisfatto da  $t$ , e per ogni criterio esiste un test che lo soddisfa. Ovvero

$$\forall c \in C . \exists t \in T \text{ tale per cui } c(t) \text{ è true} \wedge \forall c \in C . \exists t \in T \text{ tale per cui } c(t) \text{ è true}.$$

I criteri possono avere delle proprietà:

### Def Affidabilità di un criterio - $reliable(P, C)$

Un criterio è affidabile per  $P$  se per ogni coppia di test  $T_1$  e  $T_2$  selezionati da  $C$ , se  $T_1$  ha successo lo ha anche  $T_2$  e viceversa. Quindi, se  $T_1$  trova un errore lo trova anche  $T_2$  e viceversa. (insomma se tutti danno lo stesso risultato)

Non è più un problema dei dati, ma del criterio che deve riuscire a selezionare dati di test che siano tutti affidabili (o tutti NON affidabili). Sposto il legame con la rilevazione dell'errore dai dati al criterio.

Non è necessario che tutti trovino lo stesso fallimento; basta che ce ne sia uno.

### Def Validità di un criterio - $valid(C, P)$

Un criterio è detto valido se quando il programma non è corretto allora è effettivamente in grado di trovare un errore.

Quindi sto spostando il legame con la rilevazione dell'errore dall'errore al programma.

### Esempio

- $selected(C, T) . T \subseteq \{0, 2\} \rightarrow reliable(double, C), \neg valid(C, double)$   
ovvero negli input che non mi danno problemi, allora sicuramente il criterio è  $reliable(double, C)$  perché seleziona solo test che mi danno la stessa risposta ("il programma è corretto). Tuttavia, non è  $valid(C, double)$ : perché il programma di fatto non è giusto e questi dati di test non mi permettono di trovare l'errore.
- $selected(C, T) . T \supset \{0, 2\} \Rightarrow \neg reliable(double, C), valid(C, double)$   
ovvero ho dentro sia input buoni che input che rilevano l'errore, allora ho  $\neg reliable(double, C)$  e  $valid(C, double)$
- $selected(C, T) . T \subseteq [3, \infty) \Rightarrow reliable(double, C), valid(C, double)$   
allora ho sia  $reliable(double, C)$  che  $valid(C, double)$

```
program double (in,out)
var x, y: integer;
begin
  read(x);
  y := xx;
  write(y);
end
```

### Teorema di Goodenough e Gerhart

$$reliable(C, P) \wedge valid(C, P) \wedge selected(C, T) \wedge \neg success(T, P) \Rightarrow ok(P)$$

E' il primo risultato su questi aspetti. Se trovo almeno un criterio di test è affidabile, valido e se almeno un test selezionato dal criterio il programma non è di successo (ovvero è  $ok(P, T)$ )... allora il programma è corretto.

Funziona perché

- Se il test è reliable, allora tutti i possibili input danno lo stesso risultato di  $T$ , ovvero not success
- $C$  è valid, quindi se ci fosse un errore lo troverebbe.

E' chiaro che sono condizioni molto forti, e in generale questo problema non è decidibile.

### Teorema di Howden

Non esiste un algoritmo che, dato qualunque programma  $P$ , genera un test ideale finito, ovvero un test selezionato da una condizione valid E reliable.

La decidibilità non ce la fa proprio per colpa di quella E.

L'intuizione della dimostrazione è che se io avessi un programma che è corretto su tutti gli input tranne che su un input  $d$ , e supponendo di poter fornire un criterio sia affidabile che valido.. vorrebbe dire che dovremmo generare solo test che contengono  $d$ , che però noi non conosciamo. Quindi dovremmo generare tutti i possibili input, che sono infiniti.

Quindi il problema di decidibilità: è possibile che esista, ma non riusciamo a calcolarlo in modo algoritmico.

## Test esaustivo

E' l'unica alternativa, ma è assolutamente impossibile – persino in contesti finiti. Se pure un programma fosse terminante, ci sono una quantità davvero esponenziale di possibili esecuzioni.

## Tesi di Dijkstra

Non è dimostrata, ma è chiara anche alla luce delle informazioni che abbiamo da Fondamenti:

I programmi di testing possono rivelare la presenza di errori nel programma, ma non possono provare la loro assenza.

Legato a questo, ci sono numerosi problemi di decidibilità:

 **Weyuker theorem**

Given any program P, the following problems are undecidable:

- There exists an input causing the execution of a particular statement?
- There exists an input causing the execution of a particular branch?
- There exists an input causing the execution of a particular path?
- Is it possible to find at least one input causing the execution of any statements in P?
- Is it possible to find at least one input causing the execution of any branch in P?
- Is it possible to find at least one input causing the execution of any path in P?

 Fine

## Generazione dei casi di test

Tutte queste informazioni ci fanno capire che la scelta dei test è cruciale per determinare la qualità del testing. Quindi, **il problema si sposta sul capire quando i nostri casi di test sono adeguati.**

In generale, quindi, il testing include una certa creatività: devo capire quello che avviene nel programma dopo averlo scritto. Generare i casi di test è difficile, perché bisogna superare i limiti imposti durante la programmazione.

**E' essenziale pianificare come scegliere i casi di test, e misurare le risposte e capire quando fermarsi.**

### Divisione in classi di equivalenza

La generazione dei casi di test passa attraverso il **partition testing**, ovvero cerco di **dividere i possibili input del programma in classi di equivalenza** e avere almeno un candidato per ogni classe.

Questo si può fare in vari modi:

- **Randomicamente:** suppongo che l'errore sia uniformemente distribuito nell'esecuzione del programma. E' il più semplice, ma non sfrutta eventuali informazioni che abbiamo sul programma.
- **Funzionale (black box):** non conosco niente del mio codice se non la funzionalità, e mi baso sui casi limite della funzione che sto calcolando. Ad esempio, se sto facendo il doppio, posso testare quei numeri in cui so che il risultato si sovrappone ad altre funzioni.
- **Strutturale (white box):** conosco il codice, e individuo gli elementi in base a questo
- **Grey box:**  


... di solito faccio un mix ...
- **Fault-based:** è abbastanza vicina alla funzionale. Ho una classe di errori che ritengo possano essere generati dal programma, e cerco di sollecitare proprio quelli.

Noi ci focalizziamo sulla strutturale.

## Divisione strutturale: code-based testing

Così come l'analisi statica cerca di approssimare semantica dal codice, così qui sfruttiamo la stessa informazione per generare casi di test e misurare se posso ritenere l'insieme generato sufficiente.

In questo caso, il termine che si utilizza è la **coverage**.

### Coverage

#### Def Coverage

E' una misura di completezza. Cerca di caratterizzare rispetto a elementi di riferimento quanto i miei test li coprano; in che percentuale l'esecuzione degli elementi considerati importanti è eseguita.

"Quali elementi non sono ancora stati eseguiti?"

Ci accertiamo che non ci siano parti di programma che il mio test non sia andato a sollecitare. Ma quali sono queste parti? Tipicamente:

- **Control flow graph**
- **Statements** (CFG nodes) o **branches** (CFG edges)
- Combinazioni e frammenti: **condizioni** o **cammini**.

Questi aspetti possono anche essere combinati con testing funzionali.

Ugualmente, comunque, non ho nessuna garanzia: **eseguire tutti gli elementi del CFG non mi assicura di aver trovato tutti gli errori**; questo perché eseguire un'operazione sbagliata non è detto che mi sia sufficiente a rendere quell'errore evidente.

L'idea è che questo strumento può alzare la confidenza che ho nel testing per **migliorare l'affidabilità** dei casi di test.

Il criterio su cui si basa il valore di coverage è il **criterio di inadeguatezza**: ovvero, si parte dall'assunzione che **se alcune parti di codice non vengono testate allora il testing non è adeguato**. Praticamente, uso la coverage come indice di quando posso considerare i miei test adeguati: finché non ho coperto una percentuale sufficiente di elementi non mi ritengo soddisfatto.

Criteri per calcolare la coverage:

- **Statement testing**: coverage dei comandi(nodi)
- **Coverage dei rami** (archi)
- **Coverage delle condizioni primitive**; cioè per ogni guardia il test controlla che la guardia sia stata testata almeno una volta a vero e almeno un'avolta a falso.
- **Coverage dei cammini**
- **Coverage di data flow**: sono legate a tutte le tecniche di analisi statica che abbiamo già visto.

L'obiettivo ovviamente è trovare un compromesso fra ciò che è impossibile e ciò che non è ancora sufficientemente dettagliato per dare una risposta.

#### Statement testing

**Criterio di adeguatezza - obiettivo**: ogni statement o node del CFG deve essere eseguito almeno una volta.

**Coverage** – misura del raggiungimento dell'obiettivo:  $\frac{\# \text{statement eseguiti}}{\# \text{statement totali}}$

**Ratio**: l'errore in un comando può essere rivelato solo eseguendolo, E' una condizione non necessaria ma sufficiente.

#### Esempio 1

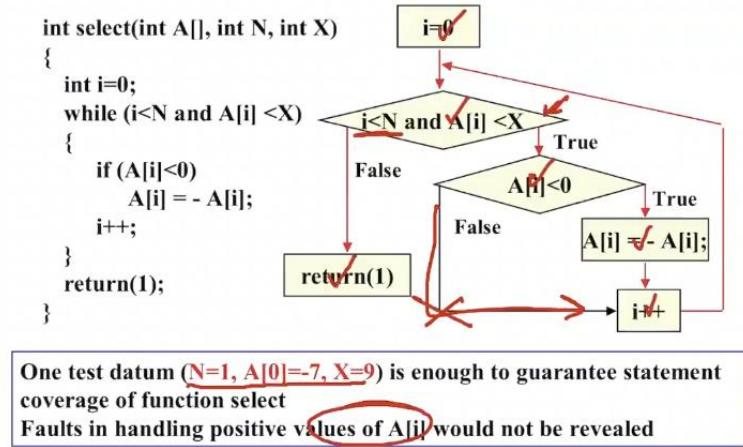
```
1  int foo (int a, int b, int c, int d, float e) {  
2      float e;  
3      if (a == 0) {  
4          return 0;  
5      }  
6      int x = 0;  
7      if ((a==b) OR ((c == d) AND bug(a) )) {  
8          x=1;  
9      }  
10     e = 1/x;  
11     return e;  
12 }
```

Ci interessa eseguire ogni comando almeno una volta.

Consideriamo l'input {0,0,0,0}. In questo modo, vengono eseguite solo le prime 5 righe. Per coprire anche le altre, posso aggiungere il test {1,1,1,1} per arrivare a fare tutte le altre istruzioni alle linee 6-12.

Prendendo entrambi questi input, prendo il 100% del programma.

*Esempio amonaolta*



E' il può grezzo: per esempio, in questo caso non viene mai percorso il ramo False del test  $A[i] < 0$

Branch testing

Criterio di adeguatezza: ciascun ramo (edge del CFG) deve essere eseguito almeno una volta.

$$\text{Coverage: } \frac{\# \text{ rami eseguiti}}{\# \text{ rami}}$$

*Esempio 1*

Vogliamo degli input per cui riesco a prendere sia il ramo vero che il ramo falso. I due test scelti in precedenza mi risolvono già il test a riga 3, perché una volta è vero e una volta è falso. Il problema invece è sulla condizione a riga 7, perché vogliamo vederla sia a vero che a falso.

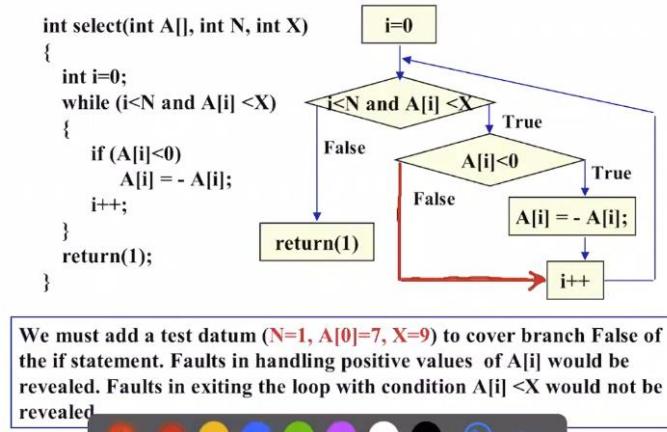
| Line # | Predicate                         | True  | False   |
|--------|-----------------------------------|---|---|
| 3      | (a == 0)                          | Test Case 1<br><code>foo(0, 0, 0, 0, 0)<br/>return 0</code> | Test Case 2<br><code>foo(1, 1, 1, 1, 1)<br/>return 1</code> |
| 7      | ((a==b) OR ((c == d) AND bug(a))) | Test Case 2<br><code>foo(1, 1, 1, 1, 1)<br/>return 1</code> |   |

Un esempio che copre il caso mancante è (1,2,1,2,1), perché in questo modo rendiamo falso sia  $a == b$  che  $c == d$ .

Raggiungere il 100% non è tipicamente realistico su programmi di grandi dimensioni. Si mette, di solito, un 50 – 85%.

Attenzione: ancora non abbiamo mai testato la condizione  $\text{bug}(a)$ .

## Esempio 2



## Condition testing

### Statement vs branch

Attraversando tutti i rami del programma sicuramente attraversiamo anche tutti i nodi, ma non il contrario. Ma potremmo comunque non aver coperto tutte le condizioni: se ho delle consizioni con dei connettivi logici, potrei aver verificato solo la prima parte e non le sottoespressioni successive.

Cerchiamo di evitare gli errori dovuti a condizioni di test che non sono state testate. Vogliamo testare le condizioni individuali in ciascuna espressione booleana.

Criterio di adeguatezza: ciascuna condizione semplice deve essere eseguita almeno una volta.

$$\text{Coverage : } \frac{\# \text{ di valori di verità presi da tutte le condizioni}}{2 * \# \text{condizioni}}$$

Possiamo soddisfare questa condizione senza soddisfare il branch coverage.

### Esempio 1

Nella linea 7 del primo esempio abbiamo già visto che ci sono tre sotto espressioni:  $a == b$ ,  $c == d$  e  $\text{bug}(a)$ . Non abbiamo mai testato il vero/falso di  $\text{bug}(a)$ .

Per poterla testare dobbiamo scoprire come funziona  $\text{bug}(a)$ , ma possiamo già osservare che abbiamo già testato...

| Predicate | True  | False  |
|-----------|---|--|
| (a==b)    | Test Case 2<br><b>foo(1, 1, x, x, 1)<br/>return value 0</b> | Test Case 3<br><b>foo(1, 2, 1, 2, 1)<br/>division by zero!</b> |
| (c==d)    |   | Test Case 3<br><b>foo(1, 2, 1, 2, 1)<br/>division by zero!</b> |
| bug(a)    |   |  |

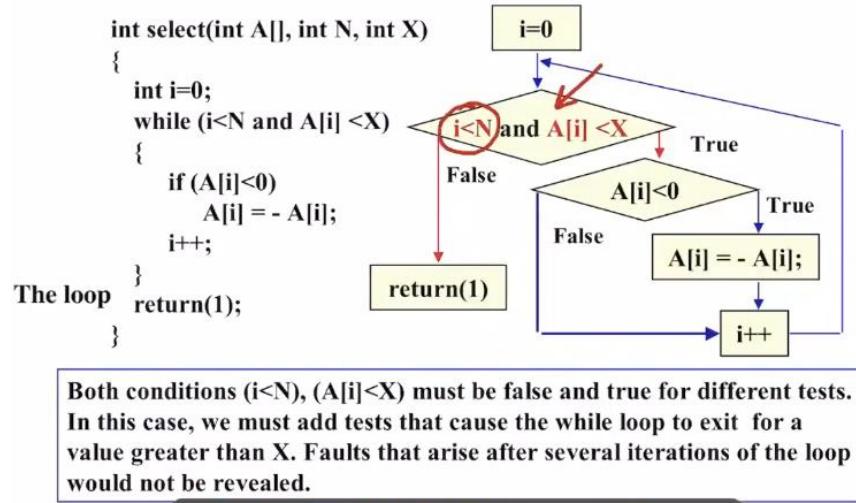
Ci manca proprio  $\text{bug}(a)$ , che è definito come "ritorna vero quando  $a = 1$ ". In questo caso, l'input che vede anche  $\text{bug}$  è

| Predicate | True  | False  |
|-----------|---|--|
| (a==b)    | Test Case 2<br><b>foo(1, 1, 1, 1, 1)<br/>return value 0</b> | Test Case 3<br><b>foo(1, 2, 1, 2, 1)<br/>division by zero!</b> |
| (c==d)    | Test Case 4<br><b>foo(1, 2, 1, 1, 1)<br/>return value 1</b> | Test Case 3<br><b>foo(1, 2, 1, 2, 1)<br/>division by zero!</b> |
| bug(a)    | Test Case 4<br><b>foo(1, 2, 1, 1, 1)<br/>return value 1</b> | Test Case 5<br><b>foo(3, 2, 1, 1, 1)<br/>division by zero!</b> |

Con questi esempi di input abbiamo una copertura al 100% anche delle condizioni.

### Esempio 2

Non abbiamo mai testato l'uscita dal loop con  $A[i]$



## Coverage dei cammini

La richiesta, ovviamente, sarebbe di esplorare tutte le possibili sequenze di cammini del control flow. E' chiaro però che se ho cicli mi esplodono i percorsi... Quindi, mi serve un compromesso.

Criterio di adeguatezza: ogni cammino deve essere eseguito

$$\text{Coverage: } \frac{\# \text{cammini eseguiti}}{\# \text{cammini}}$$

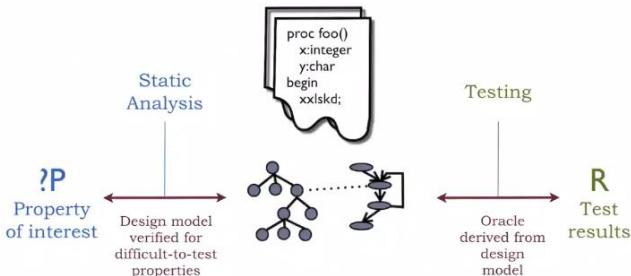
Un esempio di criterio di adeguatezza adattato ai loop è che eseguo il loop solo un certo numero di volte. E già così mi esplode! (path che esegue 0 volte, 1 volta, 2 volte, etc.)

La tecnica principale per rendere pratica la copertura dei cammini è quella di partizionare l'insieme infinito di cammini in un insieme di classi equivalenti. Alcuni criteri sono:

- # iterazioni del ciclo
- Lunghezza dei cammini da attraversare
- Dipendenze fra i cammini (selezione delle condizioni. What?)

Una coverage del 100% corrisponderebbe praticamente ad eseguire il programma.

## Combinare analisi e test



Testing for conformance to a verified design model can be more effective than directly testing for a property of interest.

Combinare analisi statica e dinamica può essere una scelta vincente. Posso sia usare la statica per aiutare il testing, sia usare la dinamica per aiutare la statica.

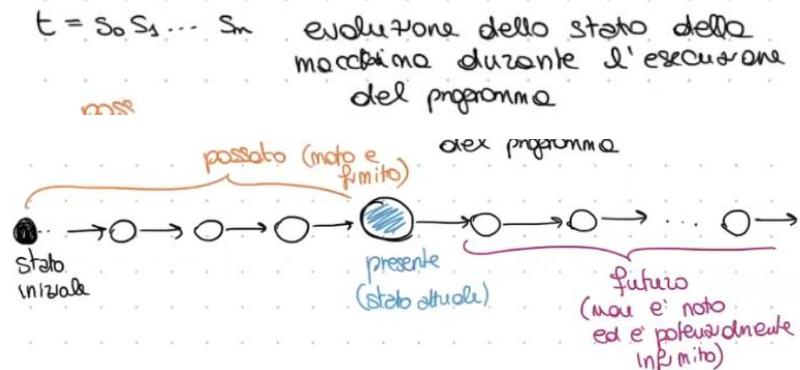
# MONITORING

## L23 - Analisi dinamica 2 - Monitoring

Il monitoring è una tipologia di analisi dinamica che permette di monitorare quello che avviene nell'esecuzione del programma rispetto a determinate specifiche. È una tipologia di analisi dinamica che segue maggiormente l'aspetto semantico.

Su cosa si basa il monitoring?

Consideriamo la semantica delle tracce. Il nostro programma è descritto attraverso la sequenza di stati del programma.



### Def Monitoring

Il monitoring consiste nel valutare la validità di una proprietà sullo stato attuale esclusivamente in funzione degli stati passati.

Sembra una banalità, ma non lo è perché questo dà forti limitazioni su quello che possiamo verificare: bisogna che la proprietà possa effettivamente essere determinata in funzione del passato...

Questo è il contesto in cui formalizziamo ciò che viene chiamato execution monitor.

### Execution monitor

Supponiamo di avere un dominio di denotazioni delle tracce come segue:

- un insieme  $\psi$  insieme di tutte le possibili sequenze di esecuzione di stati.
- $S$  sistema
- $\Sigma$  insieme di tutte le possibili tracce di esecuzione di  $S$ , e quindi  $\Sigma \subseteq \psi$

Ora possiamo definire le proprietà.

### Def Politica di sicurezza $p$

Predicato sull'insieme delle esecuzioni del sistema. In particolare,  $S$  soddisfa  $p$  se  $p(\Sigma)$  è vero.

Questo però è troppo generico per poter poi determinare se esiste un monitor che può verificare la politica. Quindi...

*Prima condizione necessaria all'esistenza del monitor: verificabilità sulle singole tracce*

La politica  $p$  è verificabile mediante un monitor se esiste un predicato  $\bar{p}$  definito sulle singole tracce tale che

$$p(\Sigma) \text{ is true} \Leftrightarrow \forall \sigma \in \Sigma. \bar{p}(\sigma) \text{ is true}$$

... ovvero è verificabile mediante un monitor se è verificabile guardando la singola traccia.

*Seconda condizione necessaria all'esistenza del monitor: deve essere una proprietà safe*

L'appartenenza di un elemento alla proprietà – ovvero la soddisfacibilità del predicato  $\bar{p}(\sigma)$  – dipende esclusivamente dall'elemento stesso  $\sigma$ . Questo si definisce come “ $p$  è una proprietà”.

Queste due proprietà sono necessarie affinché una proprietà sia monitorabile. Quindi, esistono anche proprietà non monitorabili. In particolare, ad esempio...

- **Il condizione non vale per le politiche di information flow.**

information flow = non esiste un flusso di informazione che viaggia dall'input all'output.

Come faccio a verificare se qualche informazione passa? Devo verificare se ci sono delle variazioni dell'input visualizzabili nel risultato. Per determinare se il valore di  $a$  ha effetto su  $b$  (= c'è flusso di informazione da  $a$  a  $b$ ) devo variare il valore di  $a$  e vedere se il valore di  $b$  è diverso. Ma questo vuol dire prendere almeno due esecuzioni che differiscono sul valore di  $a$  e vedere cosa succede al valore di  $b$ ; automaticamente, se devo confrontare due esecuzioni diverse, significa che la veridicità della proprietà non dipende dalla singola esecuzione. E quindi non è una proprietà.

- **Alcune politiche dipendono anche da cosa può avvenire nel futuro.**

Ad esempio, se ho un vincolo di confidenzialità su un certo documento, il fatto che sia confidenziale può essere vera in un punto ma falso in un punto futuro; non è quindi monitorabile.

## Proprietà di safety

La caratteristica formale che caratterizza queste proprietà si riassume nella proprietà di safety. Intuitivamente, questo significa che posso garantire che non può accadere nulla di negativo.

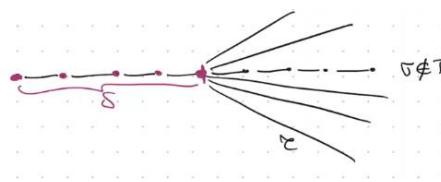
Formalmente:

$\Gamma$  è una proprietà di safety se  $\forall \sigma \in \Sigma$  dipende esclusivamente dal passato, ovvero

$$\sigma \notin \Gamma \Rightarrow \exists \delta \leq \sigma . \forall \tau \in \psi . \delta\tau \notin \Gamma$$

Con  $\delta$  prefisso di  $\sigma$ . "Sembra più complicata di quello che è."

Vuol dire che se ho la mia traccia  $\sigma$  che non soddisfa la proprietà – ovvero  $\sigma \notin \Gamma$  – allora esiste un prefisso  $\delta$  (= un pezzo che va dallo stato iniziale fino a un certo punto) tale per cui qualunque sia il modo  $\tau$  con cui io completo l'esecuzione, comunque quello che ottengo non sta in  $\Gamma$ .

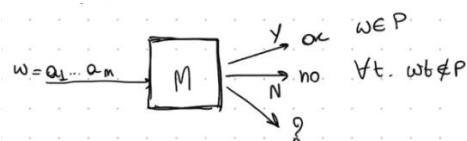


Possiamo distinguere le politiche di sicurezza in tre macrocategorie:

- **Politiche di controllo degli accessi:** proprietà di safety perché io vado a bloccare definitivamente quando c'è la violazione. Se qualcuno è entrato dove non poteva entrare, non si torna indietro...
- **Proprietà di flusso di informazione:** come visto prima non sono di safety, ma non sono nemmeno proprietà! Non dipendono da un pezzo della storia.
- **Disponibilità di una risorsa:** dipende da una sola esecuzione, ma dipende anche dal futuro. Non posso violare la proprietà in modo definitivo.

## Monitor (formale)

Un monitor  $M$  per una proprietà  $P$  è un programma che riceve in input una sequenza di stati (uno alla volta) e in funzione di questi restituisce il valore successivo se la proprietà è verificata.



## Problema del monitoring

Data una logica  $L$  (che mi serve per definire la logica dei miei predicati)

Data una formula  $f \in Formule(L)$ , cioè  $f$  descrive in modo formale la proprietà da verificare

Allora costruire un monitor per  $f$  significa costruire l'oggetto  $M_{L(f)}$  tale che se  $f$  è soddisfatta l'esecuzione del sistema continua, altrimenti agisce in qualche modo (in base al monitor).

## Enforcing

Quindi il monitor non è proprio una semplice analisi: durante l'analisi possiamo agire sull'esecuzione!

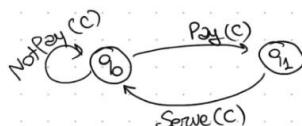
Nell'analisi ci limitiamo ad osservare, mentre qui abbiamo modificato la semantica.

Si parla di enforcing: se la proprietà è soddisfatta, allora la semantica rimane la stessa; se invece non è rispettata, la "forzo".

I linguaggi che si possono utilizzare, tipicamente, spaziano:

$$L \in \{ \text{Espressioni regolari, Grammatiche, Logiche temporali, ...} \}$$

Vediamo questo esempio come automa di una macchinetta che serve le merendine:



Con, quindi, le seguenti transizioni:

$$\left\{ \begin{array}{l} \text{notPay}(c) \wedge \text{stato} = q_0 \rightarrow \text{stato} = q_1 \\ \text{Pay}(c) \wedge \text{stato} = q_0 \rightarrow \text{stato} = q_1 \\ \text{stato} = q_1 \wedge \text{Serve}(c) \rightarrow \text{stato} = q_0 \end{array} \right.$$

Per esempio potremmo voler garantire che dopo un pagamento ci sia sempre un servizio.

## Problettiche della progettazione di un monitor

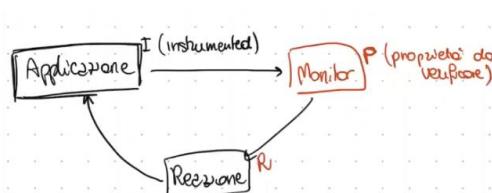
- **Instrumentazione del codice**

ovvero arricchire il codice del programma trasformandolo in un nuovo programma che ha al suo interno dei nuovi strumenti che permettono al monitor di catturare elementi. Diamo al monitor i punti di osservazione. Assomiglia ai breakpoint del debugger.

- **Scelta adeguata del linguaggio di specifica**

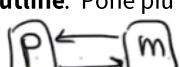
deve essere un linguaggio succinto, perché il monitor lavora insieme al programma e di conseguenza non possiamo bloccare il programma... oltre che facile ed espressivo.

Quindi, la nostra situazione è



Il livello di integrazione ("reazione") può essere fatto offline o online:

- **Offline:** è un monitor che viene applicato solamente sulla fine dell'esecuzione, ad esempio sui file di log. E' sempre possibile e non richiede proprietà di safety, però non è il monitor tipico. Non permette di evitare che avvenga la violazione.
- **Online:** è un programma eseguito parallelamente all'esecuzione del programma monitorato. Ci sono vari tipi:
  1. **Outline:** Pone più peso sull'interazione monitor, rendendo l'strumentazione meno rilevante



2. **Inline:** Si intona all'strumentazione, integrando il monitor nel programma stesso.

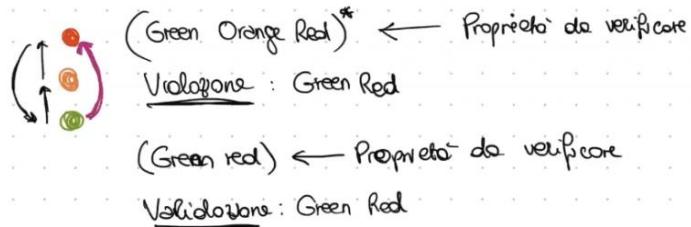


## Validazione vs violazione

Un altro aspetto importante è la differenza fra validazione e violazione. Tipicamente, il monitor si porta nel contesto della violazione: quando la proprietà è violata viene riportato. La validazione, che invece è tipica dell'analisi statica, consiste nel confermare che una proprietà vale su tutte le tracce, aka l'assenza del controesempio.

- **Validazione:** dimostrare che qualcosa di negativo non avviene
- **Violazione:** dimostrazione che è avvenuto qualcosa di negativo.

Se ad esempio consideriamo il semaforo



Validazione sarebbe che andiamo a verificare che il controesempio si verifichi.

Il monitor, tipicamente, quando trova una violazione agisce in qualche modo. Il monitor ha come obiettivo quello di evitare la violazione, e attua un enforcement della proprietà. AKA: se vale non fa nulla, se no nvale fa in modo che valga.

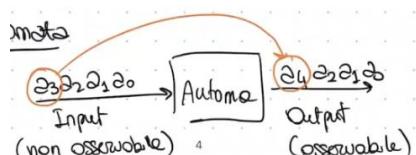
## Enforcement

Possibili reazioni:

- **Accept:** accettare significa che l'azione non viola la proprietà.  
Arriviamo ad eseguire un certo pezzo di programma che viola la proprietà, ma lo accetto.
- **Halt:** blocca il sistema  
La proprietà verrebbe violata se io consentissi l'esecuzione, quindi cambia la semantica del programma fermandolo prima che la violi.
- **Suppress:** sopprimere l'azione che avrebbe violato la proprietà  
Ovvero vado avanti ma non eseguo quella specifica azione. Devo essere sicuro che il risultato poi sia accettabile.
- **Inserire:** inserisce delle nuove computazioni che mitigano le violazioni.  
Sostituisce l'azione con altre azioni.

## Edit automata

Rappresentano le varie operazioni di enforcement. E' un modello di automa che in funzione di una traccia di input non osservabile, posso osservare la sequenza in output.



Es. sostituzione:

## Proprietà del monitor

- **Soundness (correttezza)**  
Se eseguo per intero a sequenza di input, allora quella in output sicuramente soddisfa la proprietà.  
Per ogni traccia di input esiste uno stato e una traccia di output tale per cui se io eseguo iteramente a partire dallo stato iniziale la mia sequenza, esiste uno stato di terminazione  $q'$  ed esiste una configurazione/sequenza finale in uscita che soddisfa la proprietà.  

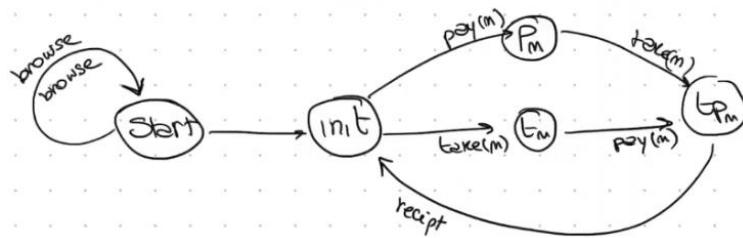
$$\forall \sigma_{in} \cdot \exists q' \exists \sigma_{out} (\sigma_{in} z q_0) \xrightarrow{*} (\text{empty}, q') \Rightarrow P(\sigma_{out})$$
- **Trasparenza**  
Se la sequenza soddisfa  $P$ , allora non viene modificata ( $=$ ) o viene modificata ma in modo equivalente ( $\cong$ ) in output.

$$P(\sigma_{in}) \Rightarrow \sigma_{in} \cong \sigma_{out}$$

Se abbiamo un monitor...

| soundness | transparent |  |
|-----------|-------------|--|
| OK        | NO          | → CONSERVATIVO   |
| OK        | OK $\equiv$ | → EFFETTUO (se si accettano alterazioni con restituzioni di sequenze valide) |
| OK        | OK $\equiv$ | → PRECISO $\equiv /$   |

Esempio



Proprietà che non vogliamo siamo sia violata: che sia rilasciata la ricevuta senza che siano state fatte le altre operazioni.

Si può combinare il monitoring con l'analisi statica.

# MODEL CHECKING

## 24 - Model Checking 1 - Strutture di Kripke

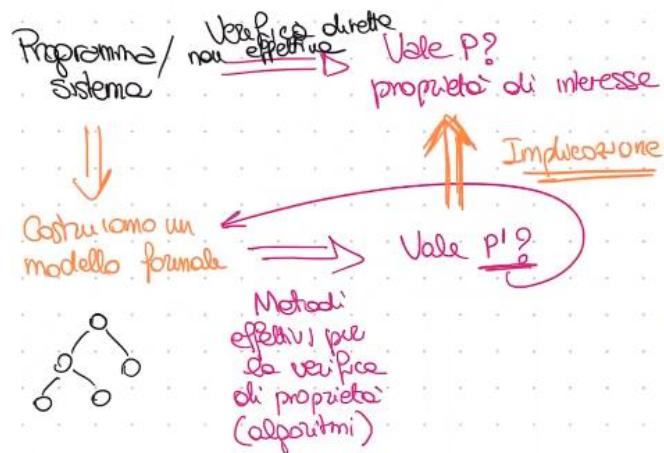
Si basa sull'idea di modellare in modo logico le proprietà che si vogliono analizzare in modo da poter applicare degli algoritmi di verifica **automatica**.

La tecnica del model checking...

- E' automatica: non serve l'interazione umana, e una volta completata la modellazione della proprietà da verificare mi basta applicare un algoritmo.
- LIMITE: non si può fare per tutti i programmi, perché opera solo su sistemi finiti. E anche su sistemi finiti può succedere che esploda.
- Corretta: sì, ma rispetto al modello
- Completa: sì, ma rispetto al modello
  - ovvero, se ho sbagliato o sono stata imprecisa nel caratterizzare formalmente il sistema reale, ho una perdita di correttezza e completezza. Ma non è dovuta al model checking in sé, ma alla costruzione errata del mio modello.
- E' statica: non richiede l'esecuzione del nostro programma.

Passaggi

Tipicamente la verifica effettiva diretta non è fattibile. Quindi costruisco un modello in base alla proprietà, e traduco la proprietà sul modello...

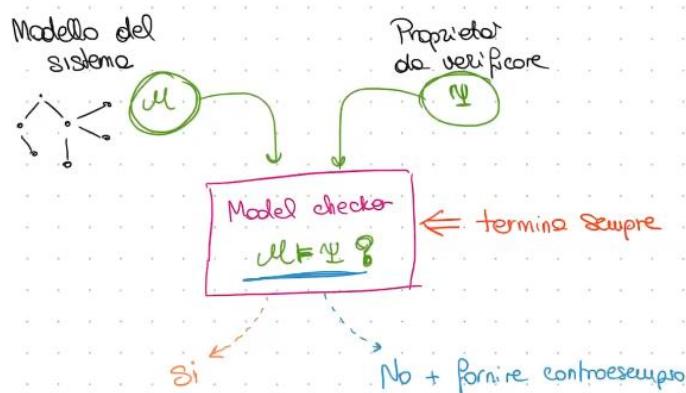


Costruire smth

Abbandoniamo il sistema reale, e costruiamo tutto sul nostro modello del sistema – che sarà un automa  $\mu$

Abbiamo anche una proprietà da verificare  $\psi$ .

Il model checker verifica se  $\mu \models \psi$ .



La risposta è positiva, oppure negativa con un controesempio.

Quindi, per applicare questo parafigma abbiamo diversi passi:

- Modellazione formale del sistema
- Specificare formalmente (linguaggi logici) la proprietà che il sistema deve soddisfare  
Intendiamo l'individuazione di una logica adeguata alla caratteristica che vogliamo catturare.
- Verificare se il modello soddisfa o meno la proprietà.  
La risposta può essere negativa se:
  1. La proprietà non è soddisfatta (nice)
  2. Modello incompleto
  3. Specifica non adeguata:  
non ho caratterizzato formalmente nel modo corretto la proprietà da verificare; il MC lavora sulla proprietà specificata, non quella che pensavo io, e quindi non va.  
Queste due, in pratica, sono legate alla distanza tra modello del sistema e sistema.
- 4. Il sistema non termina:  
il MC funziona solo su sistemi terminanti, quindi se il sistema originale è possibilmente terminante, il modello cattura solo comportamenti di terminazione.

Bisogna trovare un modello che rappresenti bene la realtà senza essere troppo complesso.

Il problema principale del model checker è l'esplosione degli stati: la loro complessità dipende molto dal numero degli stati, quindi se esplodono gli stati l'algoritmo diventa non pratico.

Un punto particolarmente interessante è l'interpretazione del risultato, in particolare in caso di risultato negativo: è importante perché voglio riuscire a capire se il problema è effettivamente del programma di partenza o è stato introdotto.

Noi vediamo le strutture di Kripke, perché è abbastanza flessibile.

Modello. Strutture di Kripke (automi a stati finiti)

Proprietà. Logiche temporali / automi

## Modellazione dei sistemi

---

Partiamo quindi dall'idea di modellare i sistemi.

Sicuramente...

- Deve descrivere stati che evolvono nel tempo
- Deve manipolare dati (variabili)
- Non determinismo: non esiste nei sistemi reali, ma sul modello ci permettono di gestire le incognite durante l'esecuzione.
- Se il sistema è un sistema a processi deve essere permessa la concorrenza. Deve gestire
  1. Sincroni/asincroni
  2. Scambio messaggi/condivisione di dati

Il modello quindi deve permettere algoritmi efficienti e deve avere strumenti per descrivere fedelmente aspetti di sistemi reali.

## Strutture di Kripke

### Definizione

La descriviamo come una tupla di un automa a stati finiti

$$M = \langle S, S_0, R, L \rangle$$

Dove

- $S$  è un insieme finito di stati
- $S_0 \subseteq S$  è l'insieme degli stati iniziali

- $R \subseteq S \times S$  è una relazione (totale) di transizione, ovvero  $\forall S \in \mathbb{S} \exists S' \in \mathbb{S}. R(S, S')$
- $L$  è una funzione di etichettatura che attribuisce label agli stati.  
 $L : \mathbb{S} \rightarrow \wp(AP)$  che associa ad ogni stato un insieme di predicati atomici.  $AP$  è l'insieme dei predicati atomici. (è ciò che distingue una struttura di Kripke da un generico automa a stati finiti).

### Def Cammino in una struttura di Kripke

Un cammino in  $M$  da uno stato  $s \in S$  è una sequenza infinita (perché non ho stati terminanti!) di stati tali che ogni coppia di stati è in relazione, ovvero

$$\pi: s_0 \dots s_n \dots$$

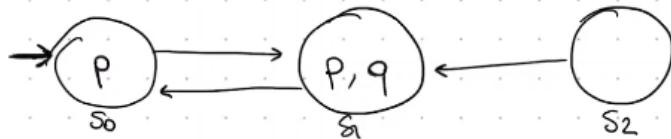
tali che  $s_0 = s$  e  $\forall i \in \mathbb{N}. R(s_i, s_{i+1})$

Per esempio

$$\begin{array}{ll} S = \{s_0, s_1, s_2\} & R = \{(s_0, s_1), (s_1, s_0), (s_2, s_1)\} \\ S_0 = \{s_0\} & L = \{s_0 \mapsto \{p\}, s_1 \mapsto \{p, q\}, s_2 \mapsto \emptyset\} \\ AP = \{p, q\} & \end{array}$$

(l'etichettatura assegna ad ogni stato quali AP soddisfa).

Tipicamente si rappresentano graficamente.



### Processo di modellazione

**Sistema  
di modello**



formuli ~~tecnico~~/descrittivo

$S_0$  formula iniziale (unica)

$R$  formula binaria di transizione

↳ traduzione

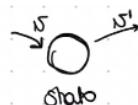
**Struttura di Kripke**

La formula di Kripke rappresenta tutte le valutazioni dei predicati che rendono veri la formula. Se una formula è vera, allora sono veri tutti i predicati che soddisfano la formula.

### Caratterizzazione delle preposizioni atomiche AP

Abbiamo:

- $V$  insieme di variabili/dati del sistema; useremo la notazione  $V' = \{\nu' | \nu \in V\}$ , ovvero copie delle variabili in  $\nu$ . In generale,  $\nu$  denota il valore precedente e  $\nu'$  il valore successivo.  
Quindi,  $\nu$  descrive il valore in ingresso allo stato e  $\nu'$  il valore in uscita.
- $D$  insieme di valori per  $V$ ; è il nostro valore di interpretazione.
- **Stato del sistema:**  $s \in S$ .  $s: V \rightarrow D$ , ovvero gli stati sono delle funzioni che associano le variabili al dominio.



### Def AP

In questo contesto,  $AP$  insieme dei predicati atomici, sono predicati del tipo

$$\nu = d \text{ con } \nu \in V, d \in D$$

Quindi le nostre proposizioni atomiche sono affermazioni di un valore attribuito alla variabile. Di fatto, questi predicati atomici nella struttura di kripke saranno

$$s \models v = d \text{ sse } s(v) = d$$

Quindi, uno stato rappresenta univocamente uno stato della nostra macchina quando le etichette corrispondono proprio ai valori che quello stato attribuisce a ogni variabile.

Quindi lo stato del modello è  $L(s) \ni \{v = d\}$

Sullo stesso stato possiamo mettere la composizione di più etichette, es

$$\nu_1 = 2 \wedge \nu_2 = 3 \wedge \nu_3 = 5$$

Poi in realtà in ogni stato dobbiamo descrivere l'intera associazione di valori alle variabili.

### Esempio

Il programma di cui vogliamo scrivere la struttura di Kripke è

$$x := (x + y) \bmod 2$$

Il primo passaggio è formalizzare queste formule, quindi supponiamo che

- Stato iniziale con entrambe le variabili a 1
- $V = \{x, y\}$
- $D = \{0, 1\}$

Quindi:

$$S_0 \equiv x = 1 \wedge y = 1 \quad (\text{stato iniziale: formula soddisfatta dai sli stati iniziali})$$

$$R \equiv y' = y \wedge x' = (x + y) \bmod 2$$

(formula che caratterizza la transizione. Perché una formula possa descrivere una transizione devo usare le variabili con l'apice. Not che

- La trasformazione non cambia y
- La trasformazione cambia x secondo la formulina matematica.

Formule  $S_0, R \rightarrow$  struttura di kripke

Vogliamo definire la formula  $M = (S, S_0, R, L)$  in funzione del mio sistema.

$S \rightarrow$  insieme di tutte le possibili valutazioni per V ( $V \rightarrow D$ )

$S_0 \subseteq S$  insieme di stati che soddisfano  $\underline{S_0}$

$\forall s, s' \in S . R(s, s') \text{ se } \forall v \in V . \underline{R}(s(v), s'(v))$   
 ↑      ↑  
 transizione da s a s'

$\forall s \in S . L(s) = \{ \underline{d} \in AP \mid s(v) = \underline{d} \}$

$R$  potrebbe non essere totale: quindi aggiungiamo la relazione riflessiva  $R(s', s)$  per ogni  $s$  che non ha successore per la  $R$ .

### Esempio

Riprendiamo l'esempio di prima

$$S_0 = x=1 \wedge y=1$$

$$R = y' = y \wedge x' = (x+y) \bmod 2$$

$$S = D \times D$$

$$\begin{matrix} \uparrow & \uparrow \\ x & y \end{matrix}$$

$S_0$  insieme di stati che soddisfano  $S_0$

$$S_0 = \{(1,1)\}$$

Per ogni stato iniziale possibile, scrivo  $x, y$  e  $x', y'$ .

$$R = \left\{ \begin{matrix} ((1,1)(0,1)) \\ x y \\ x' y' \end{matrix}, \begin{matrix} ((1,0)(1,0)) \\ x y \\ x' y' \end{matrix}, \begin{matrix} ((0,1)(1,1)) \\ x y \\ x' y' \end{matrix}, \begin{matrix} ((0,0)(0,0)) \\ x y \\ x' y' \end{matrix} \right.$$

Non devo aggiungere nulla perché tutti gli stati transiscono in qualcosa. Mi mancano le etichette per ogni stato possibile:

$$\begin{aligned} L(11) &= \{x=1, y=1\} & L(10) &= \{x=1, y=0\} & L(01) &= \{x=0, y=1\} \\ L(00) &= \{x=0, y=0\} & & & & \end{aligned}$$

Infine, posso disegnarlo. Ogni stato è un nodo, la relazione mi dà le frecce, e posso eliminare gli stati non raggiungibili.



Ho ottenuto la struttura di Kripke. 😊😊😊😊

L'unico cammino possibile è

$$\pi = (11) \rightarrow (01) \rightarrow (11) \rightarrow (01) \dots$$

### Granularità del modello

Quanto in dettaglio descrivo la transizione? Questo può interferire sulla capacità di scrivere proprietà in modo adeguato. Lo vediamo ad alto livello.

Prendiamo un sistema del tipo

$$S_0 = x=1 \wedge y=2$$

$$\begin{aligned} V &= \{x, y\} \\ D &= \mathbb{N} \end{aligned}$$

$$R(x,y) = x' = x + y \wedge y' = y \quad \alpha$$

$$R(x,y) = x' = x \wedge y' = x + y \quad \beta$$

Qui c'è del non determinismo: non ho modo di dire cosa applico prima...

- Se applico  $\alpha \rightarrow \beta$  ottengo

$$\alpha \beta \rightsquigarrow (1,2) \xrightarrow{\alpha} (3,2) \xrightarrow{\beta} (3,5)$$

- Al contrario invece

$$\beta \alpha \rightsquigarrow (1,2) \xrightarrow{\beta} (1,3) \xrightarrow{\alpha} (4,3)$$

Potrei aumentare la granularità della relazione aumentando le variabili: mi salvo da una parte i valori delle variabili e poi li uso per calcolare la somma (?) per non usare il valore alterato, e alla fine ritorno i valori su x e y.

$$\text{d0 } R(x y R_1 R_2) \equiv R_1^1 = x \quad R(x y R_1 R_2) \equiv R_2^1 = y \text{ Bo}$$

$$\text{d1 } R(x y R_1 R_2) \equiv x^1 = \underline{R_1 + y} \quad R(x y R_1 R_2) \equiv y^1 = R_2 + x \text{ Bi}$$

$$\text{d2 } R(x y R_1 R_2) \equiv x^1 = R_1 \quad R(x y R_1 R_2) \equiv y^1 = R_2 \text{ Bz}$$

E quindi se per esempio svolgiamo la sequenza di transazioni...

$$\begin{aligned} \alpha_0 \beta_0 \alpha_1 \beta_1 \alpha_2 \beta_2 \Rightarrow (1,2, \underbrace{R_1, R_2}) &\xrightarrow{\alpha_0} (1,2,1,R_2) \xrightarrow{\beta_0} (1,2,1,2) \\ &\xrightarrow{\alpha_1} (1,2,3,2) \\ &\xrightarrow{\beta_1} (1,2,3,3) \\ &\xrightarrow{\alpha_2} (3,2,3,3) \\ &\xrightarrow{\beta_2} (3,3,3,3) \end{aligned}$$

all'inizio  
possono  
essere qualsiasi  
valore

..che è un risultato ancora diverso. Quindi il modello può fare la differenza.

## MODEL CHECKING

### 25 - Model Checking 2 - Logiche temporali e algoritmo

Abbiamo detto che la struttura di Kripke è un automa del tipo

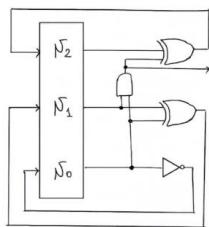
$$M = \langle S, S_0, R, L \rangle$$

Per costruire la struttura di Kripke è sufficiente fornire una formula iniziale  $S$  che dice cosa vale nel primo stato e una formula logica  $R$  che mi dice cosa vale durante la transizione. Da questo si deriva la struttura.

Questo procedimento si può applicare indipendentemente dal tipo di sistema.

*Esempio: circuiti logici*

Forniamo la regola di transizione...



E costruiamo la formula di kripke.

$$V = \{N_0, N_1, N_2\}$$

$R$ :

$$\begin{cases} N_0' \Leftrightarrow \neg N_0 \\ N_1' \Leftrightarrow N_0 \oplus N_1 \\ N_2' \Leftrightarrow (N_0 \wedge N_1) \oplus N_2 \end{cases}$$

### 1. Costruire la formula da un programma

Analogamente, possiamo costruire le formule a partire dai programmi

$$\text{Program } P \rightsquigarrow \text{Formula } R$$

1. Trasformo il programma aggiungendo le etichette prima di ogni istruzione.  $P \rightsquigarrow P^L$

Si può definire in modo induttivo:

1. Scompongo il programma in base ai punti e virgola e metto un'etichetta dopo ogni comando

$$\text{if } P = P_1; P_2 : P^L = P_1^L ; \underline{e^L} : P_2^L$$

2. Se ho un if, metto un'etichetta prima di ogni ramo

$$\text{if } P = \text{if } b \text{ then } P_1 \text{ else } P_2 \text{ endif:}$$

$$P^L = \text{if } b \text{ then } e_1 : P_1^L \text{ else } e_2 : P_2^L$$

3. Se ho un while, metto l'etichetta su un corpo

$$\text{if } P = \text{while } b \text{ do } P_1 \text{ endwhile :}$$

$$P^L = \text{while } b \text{ do } e_1 : P_1^L \text{ endwhile}$$

4. Definisco una notazione

$$\text{same}(Y) = \bigwedge_{y \in Y} (y^L = y)$$

Che mi dice che tutte le variabili dentro  $y$  sono lasciate inalterate.

2. Definisco una formula  $S_0$  in funzione delle variabili e del program counter.

La formula sarà un and fra le condizioni iniziali del programma, che dipendono dagli stati iniziali, e una condizione iniziale sul program counter che mi dice che partiamo dalla prima istruzione.

en variables  
del programma S<sub>0</sub>(V, pc) = pre(V) and pc=m yey  
program  
counter  
(vario sulle  
etichette) Condition on the initial values  
of the variables  
 (praticamente metto m come prima etichetta e m' come ultima)

3. Definisco una procedura di traduzione del programma etichettato in una formula logica, in maniera ricorsiva.

$$C(l, P, l')$$

- Assegnamento:  $l: v := e; l'$

Il program counter è  $l$  prima dell'esecuzione, e  $l'$  dopo.

$v$  dopo l'esecuzione è uguale al valore di  $e$ , e tutte le altre variabili sono lasciate inalterate.

$$C(l, v:=e, l') \equiv pc=l \text{ and } pc'=l' \text{ and } v=e$$

$$\text{same}(V \setminus \{v\})$$

- Skip

Porto avanti il PC e tutto il resto rimane inalterato.

$$C(l, \text{skip}, l') \equiv pc=l \text{ and } pc'=l' \text{ and } \text{same}(V)$$

- Composizione sequenziale:

Traduciamo tutta la prima parte di programma e mettiamo la formula in OR con il secondo pezzo di programma. E' il PC che mi fa scegliere nell'OR quale parte considerare; per ogni valore del program counter prendo solo la formula in cui il PC viene giusto.

$$C(l, P_1; l'; P_2, l') \equiv C(l, P_1, l') \text{ or } C(l', P_2, l')$$

- Condizionale:

Anche qui il PC deve essere esattamente il valore nella guardia, e a seconda del valore della guardia abbiamo un PC di uscita diverso.

$C(l, \text{if } b \text{ then } l_1: P_1 \text{ else } l_2: P_2 \text{ endif}, l')$  is disjunction of

- $\underbrace{pc=l \text{ and } pc'=l_1 \text{ and } b}_{\text{and}} \text{ same}(V) \leftarrow \text{scelta del ramo true}$
- $\text{or } \underbrace{pc=l \text{ and } pc'=l_2 \text{ and } \neg b}_{\text{and}} \text{ same}(V) \leftarrow \text{scelta del ramo false}$
- $\text{or } \underbrace{C(l_1, P_1, l')}_{\text{or}}$
- $\text{or } \underbrace{C(l_2, P_2, l')}_{\text{or}}$

- While

$C(l, \text{while } b \text{ do } l'; \text{endw}, l')$  is disjunction of

- $\underbrace{pc=l \text{ and } pc'=l_1 \text{ and } b}_{\text{and}} \text{ same}(V)$
- $\text{or } \underbrace{pc=l \text{ and } pc'=l' \text{ and } \neg b}_{\text{and}} \text{ same}(V)$
- $\text{or } \underbrace{C(l_1, P_1, l)}_{\text{or}}$

Esempio

$P = x:=e; \text{while } x < 10 \text{ do } x:=x+2 \text{ endw};$

1. Metto le etichette  $m$  e  $m'$  a inizio e fine

$P^L = m: x:=3; 1: \text{while } x < 10 \text{ do } 2: x:=x+2 \text{ endw; } m'$

2. Trasformo la condizione iniziale in unità atomiche: metto in and il pc iniziale (che deve essere  $m$ ) e le condizioni vere in quel pc.

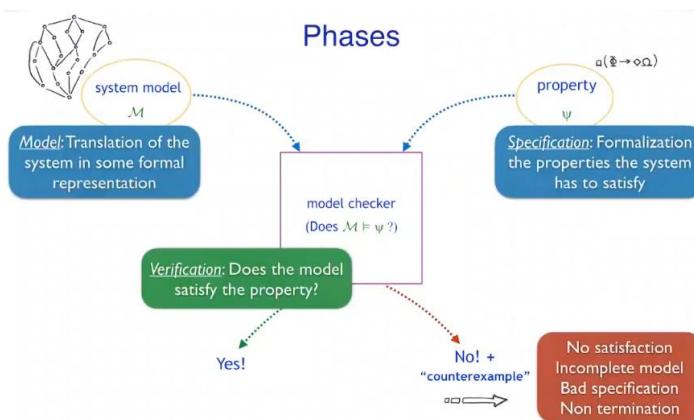
$S_0(p, x) = \underline{pc=m} \text{ and } x=0$

3. Induttivamente, la prima cosa da applicare è la composizione sequenziale: costruiamo la formula, che è la traduzione dei due sottopezzi messi in OR.  
scomponiamo tutti i pezzetti che servono

$$\begin{aligned}
 C(m, P^L, m') &= C(m, x:=3, 1) \text{ or } C(1, \text{while } x < 10 \text{ do } 2: x:=x+2, m') \\
 &= (pc=m, pc'=1, x=3) \text{ or } (pc=1, pc'=2, x < 10) \text{ or } \\
 &\quad (pc=1, pc'=m', x < 10) \text{ or } C(2, x:=x+2, 1) \\
 &= (pc=m, pc'=1, x=3) \text{ or } (pc=1, pc'=2, x < 10) \text{ or } \\
 &\quad (pc=1, pc'=m', \neg x < 10) \text{ or } (pc=2, pc'=1, x=x+2)
 \end{aligned}$$

## Logiche temporali

La situazione ad ora:



Nel model checking, queste proprietà sono importanti perché il sistema è qualcosa che evolve!

### Def. Logiche temporali

Sono dei **formalismi logici** che mi permettono di **descrivere relazioni/proprietà fra stati di sistemi cosiddetti reattivi**, ovvero che evolvono in funzione di quel che avviene nel contesto.

La differenza fra logica proposizionale e temporale sta negli operatori. Noi vedremo CTL\*, dove introduciamo degli operatori che mi permettono di descrivere le proprietà come proprietà di alberi di computazione. La radice è lo stato iniziale della struttura, e i rami sono cammini sulla struttura.

CTL\* contiene:

- **Operatori temporali:** esprimono le proprietà che riguardano l'evoluzione del cammino.
- **Quantificatori**  $\forall \exists$ , come quelli che conosciamo dalla logica, ma applicati ai cammini.

### Quantificatori

Usiamo i quantificatori.

- $A f \rightarrow$  significa che la formula  $f$  vale su tutti i cammini.
- $E f \rightarrow$  significa che esiste un cammino su cui la formula  $f$  vale.

Descrivono **proprietà dell'insieme dei cammini** che partono dallo stato.

**Parla di cammini su uno stato.**

### Operatori temporali

Sono del tutti nuovi!

- $Xf \rightarrow next | f$  vale nel prossimo stato
- $Ff \rightarrow future | f$  vale nel futuro, ovvero  $\exists$  uno stato futuro nel cammino in cui  $f$  vale.
- $Gf \rightarrow globally | f$  vale in tutti gli stati del cammino.

**Parla di stati su un cammino.**

- $f U g \rightarrow until | f$  vale in tutti gli stati finché  $g$  diventa vero.
- $f R g \rightarrow duale logico di U$

## Formule di CTL\*

Abbiamo due modi per descrivere le proprietà, e le formule saranno combinazione dei due modi: quello sui cammini e quello sugli stati.

- **Formule di stato:** descrivono proprietà dello stato (in funzione dell'insieme dei cammini che partono dallo stato)
- **Formule di cammino:** descrivono proprietà del cammino. (in funzione degli stati del cammino)

Proposizioni atomiche

Usiamo **AP** per indicare l'insieme delle proposizioni atomiche.

Le definiamo induttivamente:

### Formule di stato

- $p \in AP$  è una formula di stato.
- Se  $f$  e  $g$  sono formule di stato, allora  $\neg f, f \wedge g, f \vee g$  sono formule di stato.
- Se  $f$  è una formula di cammino allora  $Ef$  e  $Af$  sono formule di stato.

### Formule di cammino

- Se  $p$  è una formula di stato, allora è una formula di cammino
- Se  $f$  e  $g$  sono formule di cammino, allora  $\neg f, f \wedge g, f \vee g, Xf, Gf, Ff, fUg, fRg$  sono formule di cammino.

## Semantica delle formule CTL\*

Definiamo CTL\* in funzione della struttura di Kripke  $M = < \mathcal{S}, \mathcal{S}_0, \mathcal{R}, R, L >$

Come notazione scriviamo

$$\mathcal{M}, s \models p \Leftrightarrow p \in L(s)$$

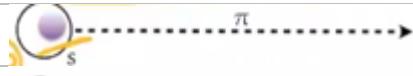
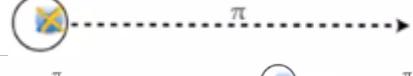
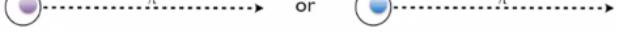
$$\mathcal{M}, s \models \neg f_1 \Leftrightarrow \mathcal{M}, s \not\models f_1$$

Scrivere che una formula soddisfa la semantica di CTL corrisponde a dire che quella formula appartiene all'insieme delle etichette.

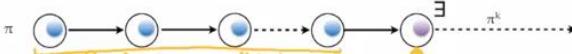
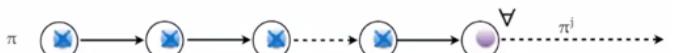
### Formule di stato

|  |  |
|--|--|
| $\mathcal{M}, s \models f_1 \vee f_2 \Leftrightarrow \mathcal{M}, s \models f_1 \text{ or } \mathcal{M}, s \models f_2$                  |  |
| $\mathcal{M}, s \models f_1 \wedge f_2 \Leftrightarrow \mathcal{M}, s \models f_1 \text{ and } \mathcal{M}, s \models f_2$               |  |
| $\mathcal{M}, s \models Eg_1 \Leftrightarrow \exists \text{ un cammino } \pi \text{ da } s \text{ tale che } \mathcal{M}, s \models g_1$ |  |
| $\mathcal{M}, s \models Ag_1 \Leftrightarrow \text{in tutti i cammini } \pi \text{ da } s, \mathcal{M}, s \models g_1$                   |  |

### Formule di cammino

|   |  |
|---|--|
| $\mathcal{M}, \pi \models f_1 \Leftrightarrow s \text{ è il primo stato di } \pi \text{ e } \mathcal{M}, s \models f_1$       |  |
| $\mathcal{M}, \pi \models \neg g_1 \Leftrightarrow \mathcal{M}, \pi \not\models f_1$  |  |
| $\mathcal{M}, \pi \models g_1 \vee g_2 \Leftrightarrow \mathcal{M}, \pi \models g_1 \text{ or } \mathcal{M}, \pi \models g_2$ |  |

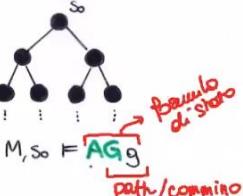
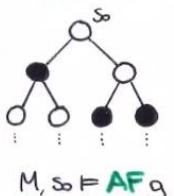
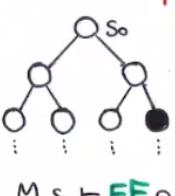
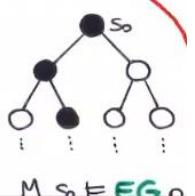
### Operatori temporali

|   |  |
|---|--|
| $\mathcal{M}, \pi \models Xg_1 \Leftrightarrow \mathcal{M}, \pi^1 \models g_1$<br>$\pi^1$ è il sottocammino di $\pi$ che parte dallo stato <b>successivo</b> a quello iniziale di $\pi$   |  |
| $\mathcal{M}, \pi \models Fg_1 \Leftrightarrow \exists k \geq 0 \text{ tale che } \pi^k \models g_1$  |  |
| $\mathcal{M}, \pi \models Gg_1 \Leftrightarrow \text{Per ogni } i \geq 0, \pi^i \models g_1$  |  |
| $\mathcal{M}, \pi \models g_1 U g_2 \Leftrightarrow \text{Esiste un } k \geq 0 \text{ tale che:}$<br>$\mathcal{M}, \pi^k \models g_2 \text{ per ogni } 0 \leq j < k \text{ e } \mathcal{M}, \pi^j \models g_1$<br>Ovvero finché $g_2$ non diventa vera vale $g_1$ |  |
| $\mathcal{M}, \pi \models g_1 R g_2 \Leftrightarrow \forall j \geq 0,$<br>$\text{se } \forall i < j \mathcal{M}, \pi^i \not\models g_1 \text{ allora } \mathcal{M}, \pi^j \models g_2$  |  |

### Esempi

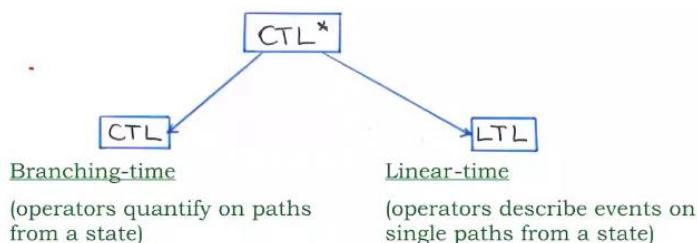
Supponiamo di rappresentare la validità del predicato annerendo lo stato.

$$g = \text{●} \models g \quad \text{○} \not\models g$$

|   |   |  |  |
|---|---|--|--|
| <br>$\mathcal{M}, s_0 \models AGg$ | <br>$\mathcal{M}, s_0 \models AFg$ | <br>$\mathcal{M}, s_0 \models EFG_0$ | <br>$\mathcal{M}, s_0 \models EG_0$ |
| Su tutti i cammini che partono da $s_0$ esiste uno stato nel futuro che soddisfa $g$                                  | Tutti i cammini che partono da quello stato soddisfano quello stato.  | Esiste un cammino da $s_0$ tale che nel futuro esiste uno stato che soddisfa $g$ .                                       | Esiste un cammino da $s_0$ su cui tutti gli stati soddisfano $g$ .   |

### Logiche

In realtà, CTL\* di solito **non viene usato direttamente** perché altrimenti gli argomenti diventano troppo pesanti; sappiamo che il model checking deve sempre terminare quindi di solito vengono usate delle sottologiche:



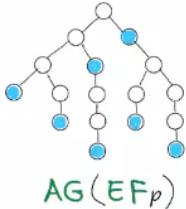
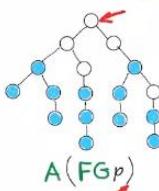
- **CTL**: prendono solo operatori che quantificano solo su cammini a partire da uno stato
- **LTL**: prendono operatori che descrivono eventi sui singoli cammini da uno stato

Poi ci sono ulteriori restrizioni, come ACTL\* e ACTL che sono le corrispondenti ma solo con quantificatori universali.

Diamo un'idea di queste restrizioni. Gli algoritmi fatti su queste restrizioni, ovviamente, sono più efficienti.

## Differenze fra CTL, LTL e CTL\*

Viene ristretta la formula di cammino:

| CTL  | LTL  | CTL*   |
|--|--|--|
| <ul style="list-style-type: none"> <li>Se <math>f</math> e <math>g</math> sono formule d'**stato**, allora <math>\neg f, f \vee g, f \wedge g, Xf, Ff, Gf, f \cup g, fRg</math> sono formule di cammino</li> </ul>   | <ul style="list-style-type: none"> <li><math>p \in AP</math>, allora <math>p</math> è una formula di **cammino**</li> <li>Se <math>f</math> e <math>g</math> sono formule di stato, allora <math>\neg f, f \vee g, f \wedge g</math> sono formule di stato</li> <li>??? allora <math>Ef, Af</math> sono formule di stato.</li> </ul> | <ul style="list-style-type: none"> <li><math>p \in AP</math>, allora <math>p</math> è una formula di stato</li> <li>Se <math>f</math> e <math>g</math> sono formule di stato, allora <math>\neg f, f \vee g, f \wedge g</math> sono formule di stato</li> <li>??? allora <math>Ef, Af</math> sono formule di stato.</li> </ul> |
| <p>Sono facili da riconoscere: abbiamo sempre <b>coppie di operatori</b> in cui ho un quantificatore universale e un operatore temporale; questo perché prima devo creare una formula di stato e poi di cammino.</p>  | <p>E' meno evidente: ho che in tutti i cammini dall'ostato iniziale deve esistere nel futuro uno stato in cui globalmente vale P. Non è CTL, perché ho l'A applicato direttamente a una formula di stato e non di cammino.</p>                      |  |
| <p>In pratica otteniamo 10 nuovi operatori di base:<br/> <math>AX, EX, AF, EF, AG, EG, AU, EU, AR, ER</math><br/> Applicando demorgan, possiamo ricavarli tutti con <math>EX, EG, EU</math></p>  |  |  |

## Model checker: algoritmo per CTL

Abbiamo il modello di sistema e la logica. Adesso ci manca il model checker, che è l'algoritmo che mi determina in modo decidibile se la formula è soddisfatta dal nostro modello.

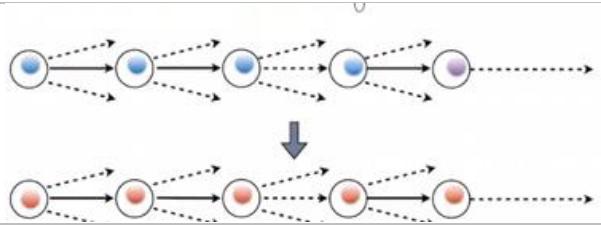
**Etichettiamo ogni stato con tutte le etichette che sono vere**, e poi processare **induttivamente** tutti gli stati successivi allo stesso modo.

### Intuizione grafica

Sto arricchendo ogni stato aggiungendo non solo i predicati atomici ma anche tutte le combinazioni di formule che valgono su quegli stati.

|  |  |
|--|--|
| <ul style="list-style-type: none"> <li>Se l'etichetta <math>f_1</math> (blue circle) non è presente in <math>s</math>, allora etichettiamo <math>s</math> con l'etichetta <math>\neg f_1</math> (red circle).</li> </ul>   |  |
| <ul style="list-style-type: none"> <li>Ugualmente, se nella formula ho <math>f_1 \vee f_2</math> (red circle), allora etichetto ogni stato che ha <math>f_1</math> (purple circle) o <math>f_2</math> (blue circle). Ad esempio, se in uno stato <math>s</math> vale <math>f_1</math>, allora gli aggiungo l'etichetta <math>f_1 \vee f_2</math>.</li> </ul> |  |
| <ul style="list-style-type: none"> <li>Se ho <math>EXf_1</math> (red circle), guardo tutti i cammini futuri: se ho almeno uno stato in cui vale <math>f_1</math> (purple circle), allora metto <math>EXf_1</math> sullo stato in cui mi trovo.</li> </ul>  |  |

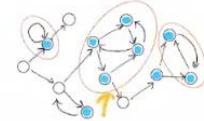
- $g = E[f_1 U f_2]$  è un po' più complesso. Devo trovare tutti gli stati etichettati con  $f_2$ . A questo punto
  1. Etichetto lo stato dove ho trovato  $f_2$
  2. Vado all'indietro a etichettare tutti gli stati in cui vale  $f_1$  con  $E[f_1 U f_2]$
- Per  $g = EGf_1$  bisogna usare un sottoalgoritmo :')



1. Restringiamo la struttura  $M$  solo sugli stati dove vale  $f_1$



2. Decompongo questa sottostruttura in tutte le componenti fortemente connesse, ovvero i sottografi in cui da ogni stato posso raggiungere tutti gli stati.  
Questi elementi vengono eliminati perché da loro [spiegazione onestamente unintelligibile. "da loro esco? What?"]
3. Tra tutti gli stati individuati fino ad ora: questi stati soddisfano  $g = EGf_1$  se :
  - Sono nella componente fortemente connessa appena individuata
  - Soddisfano  $f_1$  **E** esiste un cammino che porta dallo stato  $s$  dentro una componente fortemente connessa.



Esiste anche un altro algoritmo che passa per la definizione di punto fisso, ma non lo vediamo.

Codice

```

procedure CheckEG( $f_1$ )
   $S' := \{ s \mid f_1 \in \text{label}(s) \}$ ;
   $SCC := \{ C \mid C \text{ is a nontrivial SCC of } S' \}$ ;
   $T := \bigcup_{C \in SCC} \{ s \mid s \in C \}$ ;
  for all  $s \in T$  do  $\text{label}(s) := \text{label}(s) \cup \{ EG f_1 \}$ ;
  while  $T \neq \emptyset$  do
    choose  $s \in T$ ;
     $T := T \setminus \{s\}$ ;
    for all  $t$  such that  $t \in S'$  and  $R(t, s)$  do
      if  $EG f_1 \notin \text{label}(t)$  then
         $\text{label}(t) := \text{label}(t) \cup \{ EG f_1 \}$ ;
         $T := T \cup \{t\}$ ;
      end if;
    end for all;
  end while;
end procedure

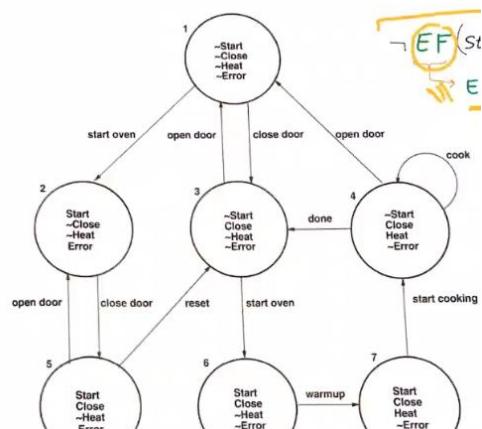
```

Esempio

$$\neg EF(Start \wedge EG \neg Heat)$$

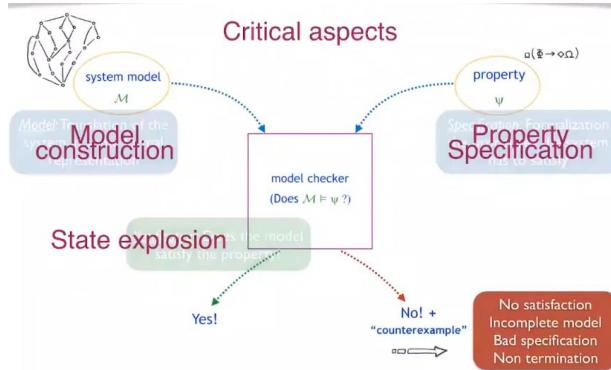
È equivalente a dire  $E[\text{true} \cup (Start \wedge EG \neg Heat)]$ .

Facciamo la verifica sul sistema forno a microonde:



|  |  |
|--|--|
|  | <p><b>Atomic formulas:</b></p> <p><math>\neg \text{Heat}</math> ●</p> <p><math>\text{Start}</math> ●</p> <p><math>\neg \text{EF}(\text{Start} \wedge \text{EG} \neg \text{Heat})</math></p> <p><math>\neg \text{E}[\text{true} \cup (\dots)]</math></p>  |
| <p><b>1. Formule atomiche</b></p> <p>Le formule atomiche della nostra formulina sono <math>\neg \text{Heat}</math> e <math>\text{Start}</math>.</p> <p>Andiamo ad etichettare tutti gli stati su cui valgono questi predicati atomici.</p>   | <p><b>2. Verifichiamo <math>\text{EG}(\neg \text{Heat})</math></b></p> <p>Prendiamo la componente fortemente connessa di stati che soddisfano <math>\neg \text{Heat}</math>, e eventuali stati esterni alla CFC che soddisfano <math>\neg \text{Heat}</math> e per i quali esiste un cammino che porta alla componente fortemente connessa (in questo caso nessuno).</p>                     |
| <p><b>3. Verifichiamo l'and fra <math>\text{Start}</math> e ciò che abbiamo verificato, ovvero <math>\text{Start} \wedge \neg \text{Heat}</math>.</b></p> <p>Quindi in pratica metto un'etichetta in tutti gli stati in cui ho sia l'etichetta di <math>\neg \text{Heat}</math> (qui arancione) che <math>\text{Start}</math> (qui verde).</p>   | <p><b>Formula:</b></p> <p><math>\text{EG} \neg \text{Heat}</math> ●</p> <p><b>SCC</b></p> <p><math>\neg \text{EF}(\text{Start} \wedge \text{EG} \neg \text{Heat})</math></p> <p><math>\neg \text{E}[\text{true} \cup (\dots)]</math></p>   |
| <p><b>4. Verifichiamo l'exists until, ovvero la nostra formula "finale" <math>\text{E}[\text{true} \cup (\text{Start} \wedge \text{EG} \neg \text{Heat})]</math></b></p> <p>Prendo tutti gli stati in cui vale la formula <math>\text{Start} \wedge \text{EG} \neg \text{Heat}</math>, che sono quelli in verde.</p> <p>Poiché <math>\text{true}</math> vale sempre, segniamo col rosso anche tutti gli stati che raggiungono gli stati etichettati.</p> | <p><b>Formula:</b></p> <p><math>\text{Start} \wedge \text{EG} \neg \text{Heat}</math> ●</p> <p><math>\neg \text{EF}(\text{Start} \wedge \text{EG} \neg \text{Heat})</math></p> <p><math>\neg \text{E}[\text{true} \cup (\dots)]</math></p> <p><b>5. Verifica della formula finale, ovvero <math>\neg \text{EF}(\dots)</math></b></p> <p>Poiché il tutto è negato, non è mai verificata 😞</p> |

## Aspetti critici



- **Costruzione del modello**: dobbiamo trovare il modo di essere fedeli al sistema reale
- **Specificà della proprietà**: dobbiamo trovare il modo di essere fedeli a ciò che vogliamo effettivamente verificare sul sistema
- **Algoritmo. Esplosione degli stati**

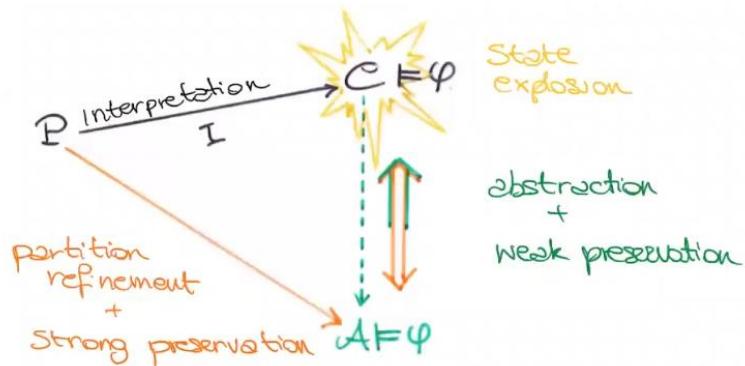
Abbiamo già visto che l'algoritmo del model checking è per definizione uno strumento che deve essere decidibile: deve sempre dare una risposta in tempi ragionevoli, rispondere sì se è sì e rispondere no con un controsenso altrimenti.

Il problema è che se ci sono troppi stati l'algoritmo non è efficiente.

L'esplosione degli stati può derivare dal fatto che abbiamo un modello troppo concreto: questo ci permette di essere molto precisi, ma aumenta il rischio di arrivare a un numero di passi di computazione non accettabile.

**Soluzione: processo di astrazione.** L'idea è di astrarre gli stati

in una struttura di Kripke astratta tale per cui si ha una weak preservation: se la proprietà è soddisfatta lo è sicuramente, ma se mi dice che non è soddisfatta e mi da un controsenso questo controsenso potrebbe non essere reale: si parla di controsenso spurio, ed è un controsenso generato dal processo di astrazione. In questo caso potrei avere un processo di raffinamento del modello che cerca di raffinare per avere una preservazione più forte.



*Concetto di preservazione*

Data  $\mathcal{A}$  astrazione di un modello  $C$ :

Si parla di **weak preservation** quando si ha **correttezza**:  $\mathcal{A} \models \varphi \Rightarrow C \models \varphi$

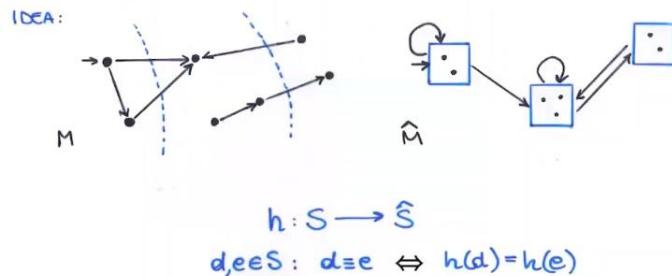
Se una formula è soddisfatta sul modello astratto, allora è soddisfatta anche sul modello concreto (ma non vale sempre l'opposto).

Si parla di **strong preservation** quando **vale anche la direzione opposta**, ovvero se non è soddisfatta nel modello astratto allora non è soddisfatta nemmeno nel sistema concreto.  $\mathcal{A} \models \varphi \Leftrightarrow C \models \varphi$

L'esplosione di stati, tipicamente, avviene con la strong preservation.

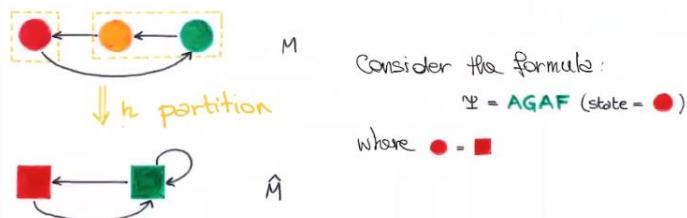
## Model checking astratto

L'intuizione è quella di astrarre gli stati: voglio accorparli in modo tale da ottenere degli stati astratti che sono insiemi di stati concreti, in modo simile a quello che facciamo in analisi statica.



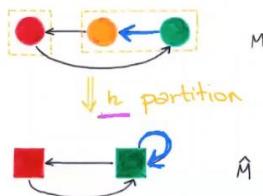
Anche la funzione di etichettatura viene semplificata accorpando etichette.

Esempio. Il semaforo



$\Psi = AGAF(\text{rosso}) = \text{ogni cammino prima o poi passa dal rosso.}$

Qui il problema è la transizione da verde ad arancione, che nell'astrazione mi porta ad avere una transizione dove rimango sul quadrato verde.



Quindi potrei rimanere per sempre nel quadrato verde! Quindi, la formula  $\Psi$  che è sempre soddisfatta nel concreto, potrebbe non esserla nell'astratto. Il controesempio che cicla per sempre nel verde è **spurio**.

while  $\hat{M} \not\models \Psi$  with counterexample

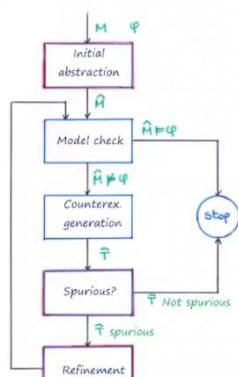


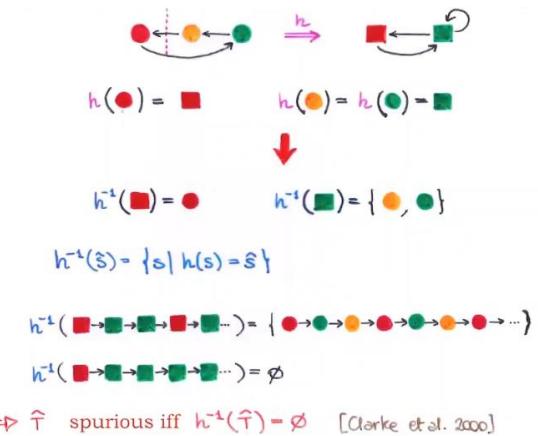
Esiste quindi un processo che aggiunge un controllo che verifica se il controesempio era spurio. →

Raffinamento del modello

1. Rilevare che un controesempio è spurio

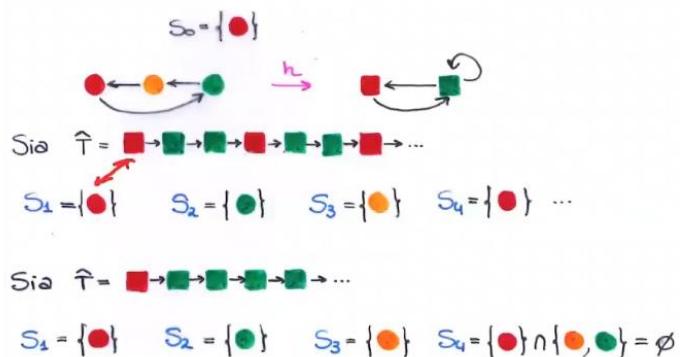
Lo vediamo a livello grafico sull'esempio del semaforo.





$h^{-1}$  è la funzione che ritorna gli stati concreti dato un elemento astratto. Se la applichiamo all'intero cammino riusciamo a costruire l'insieme dei cammini concreti che corrispondono a quello astratto: noto che il cammino del controsenso non ha un corrispondente reale.

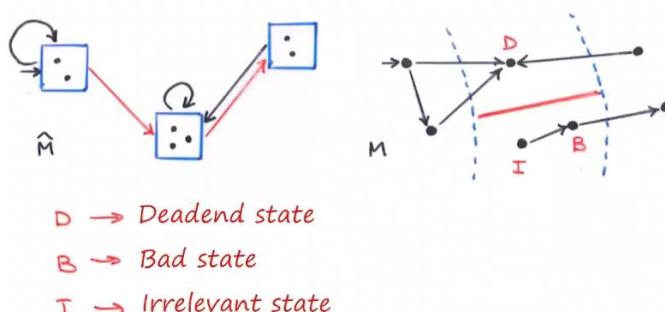
Quindi, possiamo raffinare il sistema costruendo una sequenza di insiemi  $s_i$  che mi dicono l'insieme di elementi concreti che possiamo raggiungere.



Questa è la procedura decidibile che stabilisce che il controsenso è spurio.

## 2. Rifinire l'astrazione

Guardo quello che posso raggiungere dagli stati concreti, quelli che escono da uno stato (???) e li separo.



Nell'esempio del semaforo, spezzeremmo il verde dall'arancione: Il rosso raggiunge il verde e esce dall'arancione, quindi dovrei spezzarli.