Domande di teoria di Analisi e Verifica del software

1. CFG

Definire cosa è un CFG e dare un esempio.

> (22/07,21/07,21/02,21/02,19/02,19/02,17/09,17/06,16/06,18/07,16/02,17/02,18/02,16/02,17/02)

Un CFG (*Control Flow Graph*) è un grafo orientato generato a partire dalla sintassi di un programma. È usato per analizzare il software e fare debugging; per esempio, permette di rilevare casi di codice morto (ovvero mai eseguito).
Più formalmente, un grafo è rappresentato come $G = (V, E)$ dove:

- V è l'insieme dei nodi, che in questo ambito corrispondono ai *basic blocks*. Un *basic block* è un segmento di codice che ha un solo punto di ingresso, un solo punto di uscita e non contiene branching interni.
- E è l'insieme degli archi, che in questo ambito corrispondono ai trasferimenti di controllo.

Per identificare i *basic blocks* si utilizza il concetto di *leader*, definendo leader un'istruzione che ha una di queste caratteristiche:

- È l'istruzione di ingresso o di uscita;
- È l'istruzione che segue un'istruzione di branching;
- È un'istruzione target di un'istruzione di branching.
  A questo punto, definiamo il *basic block* come l'insieme di istruzioni che va da un *leader* (incluso) al *leader* successivo (escluso).

Quindi, la procedura compreta per costruire un *CFG* è:

1. Se sono presenti più nodi di uscita o più nodi di ingresso, li riassumo aggiungendo un supernodo.
2. Individuo le istruzioni leader e con esse costruisco i basic block
3. Collego i basic block con gli archi, facendo attenzione alle operazioni di branching

Alcuni esempi di terminologia collegata ai *CFG* sono:

- $Pred(n)$ è l'insieme dei nodi che hanno un arco uscente che termina in n;
- $Succ(n)$ è l'insieme dei nodi che hanno un arco entrante che parte da n;
- *Nodo di branch*: è un nodo che ha più archi uscenti
- *Nodo di join*: è un nodo che ha più archi entranti
- *Nodo dominatore*: preso un nodo n, diciamo che domina su m se tutti i cammini che arrivano al

nodo m passano dal nodo n.

> (esempio grafico)

2. Semantica

Definire il concetto di collecting semantics e in che modo astrae la semantica operazionale delle tracce.

> (22/07,21/02,19/02,19/02,17/06,18/07,16/02,17/02,18/02,16/02,17/02)

Definire il concetto di semantica. In particolare, descrivere cosa è la semantica delle tracce e cosa è la semantica collecting. Che relazione esiste tra le due semantiche? Se volete dare degli esempi notate che una traccia si può scrivere come 1 → 2 → 3 → 4 e un insieme si può rappresentare come {1, 2}.

> (21/07,21/02)

La semantica di un programma è la **definizione del suo comportamento durante l'esecuzione**, generalmente in maniera formale. Due esempi di semantica sono:

- **Semantica denotazionale**: è un tipo di semantica che descrive il programma come una funzione matematica $Semantica : stati\ di\ input \times stati\ di\ output$, ovvero che descrive il comportamento del programma come l'insieme delle modifiche che esso effettua sulla memoria nella sua esecuzione.
- **Semantica operazionale**: è un tipo di semantica che descrive l'esecuzione del programma attraverso un sistema di transizione, mostrando gli effetti del programma ad ogni passo di esecuzione.

La **semantica delle tracce** è la rappresentazione della semantica di un programma attraverso l'enumerazione di tutte le tracce possibili, dove con tracce intendiamo una sequenza di stati raggiunti. Consiste, quindi, in un **insieme di tracce**. Questo tipo di rappresentazione, evidentemente, è molto soggetto alla non terminazione: la maggior parte dei programmi ha infinite esecuzioni possibili (per esempio, un `while` infinito, oppure anche solo una variabile di tipo `int` che può assumere qualunque valore).

La **semantica collecting** è la rappresentazione della semantica di un programma come **traccia di insiemi**: raccoglie, per ogni passo di computazione, tutti gli stati raggiungibili in quel numero di passi e genera così una traccia (sequenza) di insiemi. Questo permette più facilmente la terminazione, ma ovviamente comporta della perdita di informazione: viene perduta l'informazione riguardante la transizione da un singolo stato a un altro, generando quindi delle tracce spurie che sono descritte dalla semantica collecting ma inesistenti sul programma reale.

Prendiamo come esempio un programma che può assumere degli stati $1, 2, 3$.

Un esempio di semantica delle tracce su questo programma, assumendo che abbia un numero (molto) finito di tracce possibili, è $(1 \to 2 \to 3), (1 \to 1 \to 3), (2 \to 2 \to 3)$.
A questo punto posso generare la semantica collecting come traccia di insiemi di stati raggiungibili ad ogni passo: in questo caso ottengo $1, 2 \to 1, 2 \to 3$. Questa rappresentazione includerebbe anche la traccia $2 \to 1 \to 3$, che però in realtà non esiste ed è quindi definita traccia spuria.

La relazione fra questi due tipi di semantica è che la semantica collecting è un'**astrazione** della semantica delle tracce.

Esiste anche un'ulteriore astrazione di questo concetto, che è la **semantica delle proprietà**: funziona esattamente come la semantica collecting, ma anziché collezionare stati in degli insiemi, trasformiamo questi insiemi in una proprietà - per esempio, se lo stato rappresenta il valore numerico di una variabile, ciascun insieme può diventare l'intervallo di tutti i valori ammessi. Questo ovviamente significa non solo aggiungere delle tracce spurie, ma anche dei valori spuri (ovvero tutti quei valori presenti nell'intervallo ma non effettivamente presenti nell'esecuzione concreta).

4. Analisi di data flow

Definire le soluzioni MOP, MFP e IDEAL descrivendo formalmente le differenze.

> (16/06,16/02,16/02)

- La soluzione ***IDEAL*** è la soluzione che calcola e **combina la proprietà da calcolare su tutti i cammini del *CFG* che vengono effettivamente percorsi** durante l'esecuzione del programma. Poiché è impossibile sapere staticamente quali cammini verranno eseguiti dinamicamente, questa soluzione non è calcolabile. Qualsiasi soluzione più piccola della *IDEAL* è meno precisa, perché sta prendendo in considerazione dei vincoli in più legati a quei cammini che non vengono realmente attraversati durante l'esecuzione; al contrario, qualsiasi soluzione più grande è sbagliata perché non sta prendendo in considerazione tutti i vincoli necessari.
- La soluzione ***MOP*** (Merge Over all Paths) è la soluzione che **calcola la proprietà per tutti i cammini del *CFG* e li unisce alla fine**, indipendentemente da se essi sono effettivamente calcolati o meno. Dato che sto prendendo in considerazione più vincoli del dovuto, è più grande della *IDEAL*: $MOP \sqsubseteq IDEAL$. In generale, tuttavia, anche questa soluzione non è calcolabile, in quanto potrei avere un numero di cammini infiniti.
- La soluzione ***MFP*** (Minimum Fixed Point) è la soluzione che anziché calcolare tutti i cammini per poi combinarli insieme, si limita a **combinare il valore della proprietà su ciascun nodo di join del *CFG***. È l'unica soluzione computabile, ed è un'approssimazione della *MOP*, quindi $MFP \sqsubseteq MOP \sqsubseteq IDEAL$. Nel caso in cui l'analisi sia distributiva, fortunatamente, il risultato della *MFP* e della *MOP* coincidono.
  Per fare un piccolo esempio, immaginiamo un semplice *CFG* con i nodi $N = start, end, a, b, c.$ e con gli archi $E = (start \to a), (start \to b), (start \to c), (a \to end), (b \to end), (c \to end)$. Dove so che il path $start \to 4 \to end$ non viene mai eseguito a runtime.

Su questo *CFG* ottengo:

- *IDEAL*: $f_{end}(f_a(f_{start})) \vee f_{end}(f_b(f_{start}))$
- *MOP*:$f_{end}(f_a(f_{start})) \vee f_{end}(f_b(f_{start})) \vee f_{end}(f_c(f_{start}))$
- *MFP* $f_{end}(f_a(f_{start}) \vee f_b(f_{start}) \vee f_c(f_{start}))$

Deinire il concetto di widening e per cosa viene utilizzato

> (16/06)

Il widening è un **operatore** che viene utilizzato per **accelerare la convergenza di un'analisi, anche a costo di perdere precisione**. Viene indicato con il simbolo $\triangledown$, ed è definito come $\triangledown : P \times P \to P$, dove P è il dominio della mia analisi.

Deve rispettare due condizioni:

- $\forall x, y. y \sqsubseteq (x \triangledown y) \vee x \sqsubseteq (x \triangledown y)$
- Data una catena di partenza $x_0 \sqsubseteq ... \sqsubseteq x_n$, posso definire degli y come $y_0 = x_0$ e $y_n = y_{n-1} \triangledown x_n$ e ottenere che $y_0 \sqsubseteq ... \sqsubseteq y_n$.

Per chiarire il concetto, vediamolo applicato all'analisi degli intervalli. L'idea è che, se notiamo un trend di espansione in uno degli estremi dell'intervallo, lo **spingiamo immediatamente all'infinito**. Quindi, definiamo il widening come

- $D \triangledown \perp = \perp \triangledown D = D$;
- $D_1(x) \triangledown D_2(x) = [l_1, u_1]_x \triangledown [l_2, u_2]_x = [l, u] con:$
  - $l = l_1$ se $l_1 < l_2$, $-\infty$ altrimenti;
  - $u = u_1$ se $u_1 > u_2$, $+\infty$ altrimenti;

Questo operatore serve per poter **convergere più velocemente**.

Descrivere l'analisi di Available Expressions. Fornire sia la corrispondente equazione di punto fisso che calcola l'analisi di available expressions, sia la semantica astratta.

> (21/02,16/02)

L'analisi di *available expression* calcola, dato un punto di programma, quali espressioni si trovano già calcolate e salvate in una variabile. Per essere disponibile, ovviamente, quella variabile non potrà essere stata sovrascritta o modificata, così come i valori coinvolti nell'espressione salvata. È molto utile nell'ambito dell'ottimizzazione dei programmi, in quanto mi può permettere di ottenere dei valori già pronti senza doverli ricalcolare ad ogni uso.

Si tratta di un'analisi *forward* e *definite*.

Fornire sia l'equazione di punto fisso che calcola l'analisi di liveness, sia la semantica astratta di Liveness.

> (22/07,19/02,19/02,17/02,17/02)

L'analisi di *liveness* è un tipo di analisi statica che analizza, in ogni punto di programma, quali variabili verranno effettivamente usate in futuro. Questa analisi può essere molto utile nell'ambito dell'ottimizzazione dei programmi, perché mi permette di individuare (ed eliminare) variabili che non verranno mai usate, oppure di liberare spazio in memoria nonappena la variabile non è più *live*.

Si tratta di un'analisi *backwards* e *possible*.

Descrivere intuitivamente l'analisi di reaching definitions. Fornire sia la corrispondente equazione di punto fisso che calcola l'anaisi di Reaching Definitions, sia la semantica astratta.

> (21/02,16/06,18/02)

Dato un punto di programma, l'analisi di *reaching definitions* calcola quali definizioni - ovvero assegnamenti - "arrivano" in quel punto di programma senza essere sovrascritte, rispondendo sia con la variabile che con il punto di programma in cui è stata generata quella definizione.

Si tratta di un'analisi *forward* e *possible*.

Descrivere in cosa consiste l'analisi propagazione delle copie e descrivere l'equazione di punto fisso per l'analisi della propagazione delle copie.

> (16/02,16/02)

L'analisi di propagazione delle copie è un tipo di analisi statica che determina in ogni punto di programma quali variabili siano copie, ovvero salvano lo stesso valore.

Si tratta di un'analisi *forward* e *definite*.

Descrivere in cosa consiste l'analisi degli intervalli (dominio e operazioni astratte). Quale è la semantica astratta della valutazioni delle espressioni?

> (16/06,16/02)

Descrivere intuitivamente l'analisi degli Intervalli. Fornire la semantica astratta con e senza widening.

> (21/07,18/07)

Analisi dinamica

Definire testing e debugging, scrivendone le principali differenze.

> (19/02,19/02,17/09,17/06,18/07,17/02,18/02,17/02)

Il *testing* e il *debugging* sono entrambe tecniche di analisi dinamica.

Il *testing* è un tipo di analisi che esegue il programma su un numero limitato di *input*, individuati secondo dei criteri di selezione, e in questo modo cerca di evidenziare casi in cui il comportamento del programma è diverso da quello voluto da specifica. Per comprendere meglio il concetto usiamo i seguenti termini:

- *Mistake*: è l'errore umano (distrazione, o anche input "errato" determinato dal test) che porta il programma a un comportamento scorretto
- *Fault*: è il difetto nel codice che fa sì che il mistake non sia gestito correttamente e dia origine a un output inaspettato
- *Failure*: è l'output scorretto generato dal fault quando sollecitato con il mistake
- Errore: è una grandezza che misura la distanza fra l'output aspettato e quello ottenuto nel concreto.

In questo contesto, quindi, il *testing* è un'analisi che si propone di individuare l'esistenza di fault attraverso un numero limitato di *input*. La parte difficile di questa analisi è proprio individuare gli insiemi di *input*, o meglio il criterio con cui selezionarli: poiché non posso eseguire tutti gli *input* possibili (essendo infiniti), devo essere certo di selezionare dei criteri sufficienti ad evidenziare i *faults* se presenti. In particolare, vorrei avere dei test (e quindi dei criteri di selezione) ideali che, se NON falliscono, mi assicurano che il programma è esente da errori. Questo purtroppo non è possibile, ma è valido il contrario: se il test ha successo - ovvero individua delle failure - allora sono certa che il programma contenga dei *faults*. Possiamo dividere il *testing* in

- *Debug testing*: ha lo scopo diretto di trovare errori;
- *Validation testing*: ha lo scopo di verificare che il codice sia accettabile secondo diverse metriche (come ad esempio la *coverage* o le politiche di accesso); anche questo, quando ha successo, trova dei *fault*.

Una evidente lacuna del testing è che, benché mi permetta di individuare la presenza di faults, esso non mi permette di individuare dove essi si trovino, né tantomeno di correggerli. A questo scopo, quindi, esiste il *debugging*: il *debugging* è un insieme di tecniche pratiche che mi consentono di individuare i fault nel codice.

Riassumendo, quindi, la differenza sostanziale è che:

- Il *testing* mi permette di individuare l'esistenza del fault e della failure, ma non dove essa si trovi;
- Il *debugging* viene impiegato per individuare la posizione del fault trovato durante il testing, e quindi poterlo correggere.

Parlare del code-based testing. Definire e confrontare tra loro i vari tipi di code

coverage.

> (21/02)

Il *code-based testing* è una tipologia di testing dove si stabilisce il grado di esaustività del *testing* guardando al codice. È utile per generare i casi di test: ritengo di aver generato dei test sufficienti quando la misura che ho scelto supera una certa soglia.
Per contesto, definiamo il *testing* come un tipo di analisi dinamica che mira a rilevare la presenza di *faults* nel codice verificando su un numero limitato di input scelti *ad hoc* attraverso un certo criterio se gli output generati sono quelli aspettati o sono delle *failure*.
Nello specifico, prendiamo in analisi il *coverage* testing: la domanda a cui vogliamo rispondere è "quali parti del codice non sono ancora state testate?"; questo ha senso in quanto se una parte di codice non è mai stata testata, evidentemente il testing non può rilevare eventuali faults che vi sono presenti.
I diversi tipi di *code coverage* differiscono su come vengono definite queste "parti di codice"; distinguiamo:

- *Statement testing*: le parti di codice sono le singole linee di codice; vogliamo verificare quale percentuale di linee di codice è stata eseguita dal mio test. Possiamo calcolarlo come $n \text{ linee eseguite}/\text{linee totali di codice}$.
- *Branch testing*: le parti di codice sono i rami; ovvero, vogliamo verificare la percentuale di ramificazioni eseguite rispetto al totale. Per esempio, se ho un `if`, voglio aver verificato sia il ramo `true` che il ramo `false`; oppure se ho uno `switch case` voglio che tutti i `case` siano stati eseguiti almeno una volta. Possiamo calcolarlo come $n \text{ rami presi}/\text{rami totali}$.
- *Condition testing*: le parti di codice sono le singole condizioni. Somiglia al *branch testing*, ma con la grossa differenza che in caso di guardie composte da più predicati logici, voglio verificare che ciascun predicato logico sia stato valutato sia a vero che a falso, tenendo conto di eventuale *lazy evaluation* del codice. Posso calcolarlo come $n \text{ condizioni testate}/2 * \text{condizioni presenti}$.
- *Path testing*: verifica la percentuale di cammini percorsi durante il testing; possono essere potenzialmente infiniti, per esempio in caso di cicli, e in questo caso posso eventualmente aggiungere condizioni ulteriori come "voglio aver eseguito ciascun ciclo almeno N volte". Lo calcoliamo come $n \text{ cammini eseguiti}/\text{cammini totali}$.

Definire il concetto di monitoring. In cosa consiste? Che caratteristiche formali deve avere una proprietà per essere monitorabile?

> (22/07,21/07,21/02,21/02,19/02,19/02,17/09,17/06,18/07,17/02,17/02)

Il *monitoring* è un tipo di analisi dinamica che consiste nel valutare la validità di una proprietà dello stato attuale esclusivamente in funzione degli stati precedenti.
Più formalmente, un monitor M per una proprietà P di un programma è un programma che riceve in *input* una sequenza di stati, uno alla volta, e restituisce lo stato successivo se la proprietà è verificata.

- *Input*: $\omega = a_\perp, a_1 ... a_n$.
- *Output*:
  - $YES\ (a_{n+1})$ se la traccia soddisfa la proprietà, ovvero $\omega \in P$;
  - NO ed eventuali comportamenti correttivi (es. *enforcing*, *accepting*, *halting*, *suppressing*, *inserting*) se la traccia non soddisfa la proprietà.
  - ? altrimenti.

Il monitor di una proprietà P può esistere solamente se sono soddisfatte le seguenti proprietà della proprietà da verificare p, sulle sequenze di esecuzione degli stati ψ, sul sistema S e sulle tracce *possibili* di esecuzione Σ, che saranno $\Sigma \in \psi$:

- **Proprietà verificabile su una singola traccia**: la proprietà p è verificabile tramite monitor M se esiste un predicato p' definito sulle singole tracce tale per cui, se tutte le tracce eseguibili soddisfano p', allora p è verificata. Ovvero: $p(\Sigma) = true \Leftrightarrow \forall \sigma \in \Sigma, p'(\sigma) = true$. Inoltre, la validità della proprietà p' deve dipendere esclusivamente da σ.
- **Proprietà di safety**: se una traccia viola la proprietà, non c'è nessuna evoluzione di questa traccia tale per cui la traccia non viola più la proprietà.

Il monitor per intero, invece, può avere due proprietà:

- **Soundness**: se seguo per intero la sequenza di input, allora quella in output soddisfa la proprietà.
- **Trasparenza**: se la sequenza soddisfa p allora in output non viene modificata, o viene modificata in modo equivalente.

E rispetto a queste proprietà possiamo distinguere monitor:

- **Conservativi**: soddisfano solo la soundness.
- **Effettivi**: soddisfano entrambe le proprietà ma con alcune restrizioni sulle sequenze valide.
- **Precisi**: soddisfano entrambe le proprietà senza restrizioni.

La verifica delle proprietà può avvenire in due modi:

- **Per validazione**: dimostro che non è avvenuto qualcosa di negativo.
- **Per violazione**: dimostro che è avvenuto qualcosa di negativo.

Infine, la tecnica del monitoring presenta numerose sfide:

- **Instrumentazione del codice**: bisogna decidere quali informazioni aggiungere al codice (e come) per permettere al monitor di agganciarvisi (per la verifica della proprietà oppure per l'*enforcing*).
- **Integrazione dei due programmi**: l'analisi può essere *online* o *offline*.
- Efficienza, scelta del linguaggio...

Slicing

> Nota: non compilo questa parte in quanto lo slicing non è stato richiesto nel mio anno.

Definire in modo formale ed intuitivo cosa è lo slicing. Descrivere le forme e i tipi di slicing.

> (21/07,21/07,21/02)

Cos'è lo slicing, ed in particolare cosa distingue uno slicing dinamico da uno statico o condizionale.

> (19/02,19/02,17/09,17/06,18/07,17/02,18/02,17/02)