3.2 Program verification
Verifica dei programmi

A. Gargantini

testing e verifica del sw 2021/22

Cosa vuol dire verifica

- Differenze con il testing
  - Testing: può solo provare la presenza di difetti
  - Verifica: può dimostrare l'assenza di difetti

- Le proprietà che proveremo vere sono garantite per ogni esecuzione
  - con delle assunzione
    - (ad esempio il computer non si rompe)
    - ma anche che la memoria non finisce, overflow …..

Metodi per provare le proprietà

- Esistono diversi metodi per provare le proprietà:
  - PROOF-based: simile alla prova della matematica, (fatta a mano) con il ragionamento
    - con un proof system
    - simile al calcolo delle sequenze
  - Semi-automatic: potrebbe richiedere l'intervento dell'uomo
  - property-oriented: proveremo singole proprietà
    - non “il programma è corretto”
  - sequential programs
    - ignoriamo la concorrenza
  - pre/post development: dovrebbe essre usata durante lo sviluppo del nostro codice

Quali sono i vantaggi?

- Documentation:
  - The specification of a program is an important component in its documentation and the process of documenting a program may raise or resolve important issues. The logical structure of the formal specification, written as a formula in a suitable logic, typically serves as a guiding principle in trying to write an implementation in which it holds.
- Time-to-market:
  - Debugging big systems during the testing phase is costly and time-consuming and local ‘fixes’ often introduce new bugs at other places. Ex- pe rience has shown that verifying programs with respect to formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase and helping in the clarification of the roles and structural aspects of system components

Vantaggi delle verifica

- Refactoring:
  - Properly specified and verified software is easier to reuse, since we have a clear specification of what it is meant to do.
- Certification audits:
  - Safety-critical computer systems – such as the control of cooling systems in nuclear power stations, or cockpits of modern aircrafts – demand that their software be specified and verified with as much rigour and formality as possible. Other programs may be commercially critical, such as accountancy software used by banks, and they should be delivered with a warranty: ag uarantee for correct performance within proper use. The proof that a program meets its specifications is indeed such a warranty.
  - Esempio IEC 62304 per medical devices

Alcuni sistemi per la verifica del codice

- Java Path finder:
  - Its primary application has been Model checking of concurrent programs, to find defects such as data races and deadlocks.
- Frama-C, VCC: A Verifier for Concurrent C (MS),
  - For C code
- …

KeY tool and JML

- KeY: http://www.key-project.org
  - environment for verification of JavaCard programs.
  - Subset of Java for smartcard applications and embedded systems.
    - Universities of Karlsruhe, Koblenz, Chalmers, 1998–
    - Beckert et al: “Verification of Object-Oriented Software: The KeYApproach”, Springer, 2007. (book)
    - Ahrendt et al: “The KeY Tool”, 2005. (paper)
    - Engel and Roth: “KeY Quicktour for JML”, 2006. (short paper)
- Specification languages: OCL and JML.
  - Original: OCL (Object Constraint Language), part of UML standard.
  - Later added: JML (Java Modeling Language).
- Logical framework:
  - Dynamic Logic (DL).
  - Successor/generalization of Hoare Logic.
- Integrated prover with interfaces to external decision procedures.
  - Simplify, CVC3, Yices, Z3.
- We will only deal with the tool’s JML interface “JMLKeY”.

Usato in pratica...

- http://www.envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/

Envisage Follow Envisage Dissemination Log in

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer

Cosa vuol dire provare che un programma è corretto

Proveremo

$$\{P\}S\{Q\}$$

- Detta tripla di Hoare
- Se P è vero prima di eseguire il programma S, allora dopo varrà Q
  - P: precondizione – vale prima dell'esecuzione
  - Q: postcondizione – vale alla fine dell'esecuzione
  - S: programma o frammento di programma
- P e Q vanno pensate e scritte

Asserzioni

- P e Q sono asserzioni
- In genere metteremo le asserzioni tra {}
- Scritte in logica proposizionale
  - Con l'uso di variabili che saranno anche nel programma

  /\ per AND
    - Esempio { x>0 /\ y = 3}

  \/ per OR

  ¬ per NOT

  -> per implica : A -> B : A implica B, cioè A è più forte di B, es: x =2 -> x >0

Esempio 1

- calcolo del massimo tra due numeri:

{true}

if x> y then max := x
else max:=y

// max è il massimo
{max ≥ x /\ max ≥ y /\ (max = x \/ max = y)}

Hoare Logic with Updates

- We will use a non-standard formulation of Hoare logic called “*Hoare Logic with Updates*".
  - With this formulation one more thing is included in the Hoare triple, namely an update which corresponds to the changes to the program state that have occurred “so far".
    - The update which does not contain any substitutions is denoted by [].
  - Atomic updates:
    - Loc:= term
      - Es. [x:= 5]
  - Diverse updates
    - in sequenza “,”
      - Es. [x:= 5, y:= x+1]
    - Posso avere anche più updates in parallelo “|”

      (dovranno essere consistenti)

Updates prima di un programma

- Davanti ad un programma mettiamo degli update che aggiornano lo stato a prima dell'esecuzione del programma:
  - [U]S
  - Applica gli update U ed esegui S
- La tripla diventa:
  - {P}[U]S{Q}
    - Correttezza di S: a partire da P, applica gli updates U, esegui S, vale Q
    - A normal Hoare triple {P}S{Q}, can easily be translated to the corresponding Hoare triple with updates, namely {P}[]S{Q}.

Rules

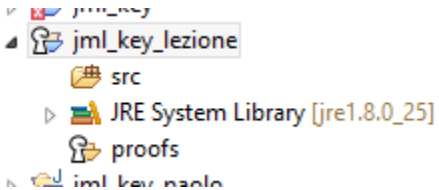
- Come per la logica anche le DL ha una serie di regole
  - che non studiamo
  - Esempio:

$$\text{Assignment}\ \frac{\{P\}[U, x:=e]\ s\ \{Q\}}{\{P\}[U]\ x:=e;\ s\ \{Q\}}$$

- cercheremo di applicarle sempre automaticamente

Come usare JML key

- Dimostrazione
- Crea nuovo progetto key
- Carica e usa Monkey



Primo esempio JMLKey

```
//@ public normal_behavior
//@ requires x == 5;
//@ assignable \nothing;
//@ ensures \result==13;
public static int assign(int x) {
   x = x + 1;
   x = x * 3;
   x = x - 5;
   return x;
}
```

Problema 1.

Alcune volte non riesco a dimostrare perché le precondizioni sono troppo deboli:

```
//@ ensures \result > 0;
 static int foo(int x){
    return x + 10;
 }
```

Devo rafforzare le precondizioni

```
//@ requires x > 19;
 //@ ensures \result > 0;
 static int foo(int x){
    return x + 10;
 }
```

Conditional

The conditional rule is applied when the first statement in the program is an if-statement.

$$\text{Conditional}\quad \frac{\{P \,\&\, U(b)\}[U]s1;s\{Q\},\ \{P \,\&\, \text{not } U(b)\}[U]s2;s\ \{Q\}}{\{P\}[U]\ \text{if (b) s1 else s2;}\ \ s\ \{Q\}}$$

The rule says that if the program starts with an if-statement, then proving the Hoare triple amounts to proving it for the program that corresponds to the then part and the program that corresponds to the else-part.

We get two new proof obligations since there are two possible paths that the program is executed, and our specification must hold in both cases. The rest of the program, s, is appended to both the then and the else-part. This is natural since in both cases the execution will continue after the if-statement.

Esempio MAX

- Nel tool

- Se non riesco a dimostrare la correttezza, devo controllare il ramo che fallisce

Correttezza di una classe

- Se voglio provare la correttezza di una classe devo provare
- operazione di creazione
  - per ogni nuova istanza INV vale
    - $\{PRE_{costr}\}$ constructor $\{INV \wedge POST_{costr}\}$
  - Il costruttore deve essere annotato
- ogni altro metodo OP

se chiamo un metodo OP con PRE vero (e INV) allora dopo l'esecuzione del metodo vale POST e continua a valere INV

$\{PRE_{op} \wedge INV\}$ OP $\{POST_{op} \wedge INV\}$

Esempio

- Counter
  - Implementa un contatore che all'inizio vale 0 e i metodi inc e dec decrementano il contatore.
  - getCounter restituisce il valore del contatore

- Scriviamo JML
  - counter è privato ma usato nelle specifiche
- Counter può andare < 0???

- Proviamo con KEY

Esempio 2

- Positive counter
  - Come prima ma che controlla che il contatore è sempre positivo (pre condizione)
- Positive with invariant
  - Aggiungi l'invariante e provalo

- Alcune volte l'invariante è necessario per provare il contratto

While cycles

Nel caso di cicli le cose di complicano

A first example

Let's look at the simplest possible example where we don't know how many times the loop will be repeated.

```
//@ requires timer >= 0;
//@ ensures \result==0;
public static int countdown(int timer)
  {
      while(timer>0){
           timer --;
      }
      return timer;
}
```

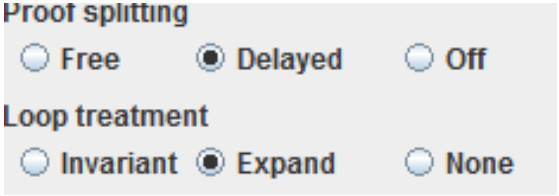
in caso di array

```
//@ requires a !=null;
//@ ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
public static void setto1(int[] a){
   int    i = 0;
   while  (i < a.length) {
     a[i] = 1;
    i++;
    }
}
```

Due modi per provare i while

- Unrolling
  - Fa diventare il ciclo una sequenza
- Funziona se il numero di iterazione è limitato
  - Ad esempio per una precondizione
- Proviamo con JMLKEY

- Se il ciclo è illimitato questo non funziona:
  - Serve un invariante




```
//@ requires timer >= 0 && timer <= 10;
//@ ensures \result==0;
public static int countdownN(int timer) { ...}
```

Loop invariant

While example

```
//@ requires timer >= 0;
//@ ensures \result==0;
public static int countdown(int n){

      while(n>0){
            n --;
      }

      return n;

}
```

- After the execution of the while loop has finished we know that n <= 0 (since the condition n > 0 evaluated to false).
- But that is not enough to conclude the postcondition, n = 0.
- Looking at the precondition, n >= 0, we see that if we also had this after the loop, then we could conclude the postcondition

(n <= 0 & n >= 0) implies n = 0.

- In order to have this, we need to show that if n >= 0 holds before the loop, then the same thing also holds after it.
- We can do this by showing that if n >= 0 holds before executing the loop body, it also holds after the loop body.

While example

{n >= 0 }

while (n > 0) n = n - 1;

{ n = 0 }

- We say that the constraint n >= 0 should be preserved when executing the loop body.
- If it does we call it a loop invariant(it doesn't vary between each repetition of the loop).
- Once we've shown that the constraint is preserved by the loop we can use arguments of induction to conclude that regardless of the number of times the loop is repeated, the loop invariant will hold after the loop if it held before it.

While statement

- Elements of proving a loop correct

1. Come up with some constraint, I , which is hopefully a loop invariant.
2. Show that I holds before the loop, ie. I is initially valid.
3. Show that I is preserved by the loop, ie. if it holds before an execution of the loop body, then it holds after. When showing this can also assume the loop condition to be true (otherwise the loop would have stopped).
4. Show that if I is valid after the loop, then after executing the rest of the program, the postcondition will be valid.
    - In other words, when proving that the rest of the program is correct we can use the invariant. Apart from using the invariant, we can also assume the loop condition to be false right after the loop (otherwise the loop would have continued).

Loop rule

$$\text{loop rule}\ \frac{P \rightarrow U(I),\ \{I \text{ and } b\}S\{I\},\ \{I \text{ and not } b\}[]s2\{Q\}}{\{P\}[U]\text{while } b \text{ do } S \text{ od } s2\{Q\}}$$

I: invariant

The rule creates three new proof obligations. As we have seen, this rule differs from the others in that it requires you to invent something new, namely the loop invariant I . I doesn't appear anywhere below the inference line, so it cannot be directly read o the current Hoare triple.

Loop rule

$$\frac{P \rightarrow U(I),\ \{I \text{ and } b\}S\{I\},\ \{I \text{ and not } b\}[]s2\{Q\}}{\{P\}[U]\text{while } b \text{ do } S \text{ od } s2\{Q\}}\ \text{loop rule}$$

- The new proof obligations are (names refer to what appears in the KeY-Hoare proof tree):
- **P → U(I)**
- Invariant Initially Valid - The invariant must hold at the beginning of the loop. This means it must be a consequence of P, hence the implication. Just as in the exit rule, the eects of the update, U, must be reflected in the consequent.

{I and b}S{I}

- Preserves Invariant - If the invariant holds before executing the loop body, it should hold afterwards. The Hoare triple has an empty update, corresponding to the fact the U was already applied before the loop, in the “Invariant Initially Valid". The changes to the program state before the loop are already reflected in I . Just as we could assume that the guard evaluated to true in the then-branch of the conditional statement, we can
- here add the same assumption when entering the loop body.

{I and not b}[]s2{Q}

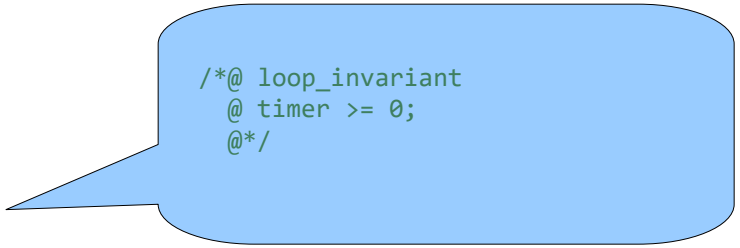
- Use invariant – By inductive reasoning we can now assume that I holds after the loop. And we also have that b is false (loop exit). These two thing form the precondition in the Hoare triple for the rest of the program

A first example

Let's look at the simplest possible example where we don't know how many times the loop will be repeated.

```
//@ requires timer >= 0;
//@ ensures \result==0;

public static int countdown(int timer) {
      while(timer>0){
            timer --;
      }
      return timer;
}
```



Diverges

- Quando usiamo un ciclo per essere sicuri che sia corretto non basta l'invariante, dobbiamo provare che il ciclo termini
- Se vogliamo evitare la prova dobbiamo ammettere che non si fermi con la notazione
  `//@ diverges true;`
- Se vogliamo inoltre precisare che il contratto si riferisce al caso in cui termini senza eccezioni aggiungiamo:
- `//@ normal_behavior;`

- `Esempio:`

```
//@ public normal_behavior
//@ requires timer >= 0;
//@ ensures \result==0;
//@ diverges true;
```

A first example

Let's look at the simplest possible example where we don't know how many times the loop will be repeated.

```
//@ public normal_behavior
//@ requires timer >= 0;
//@ ensures \result==0;
//@ diverges true;
public static int countdown(int timer)
  {
      while(timer>0){
           timer --;
      }
      return timer;
}
```

in caso di array

```
//@ public normal_behavior
//@ requires a !=null;
//@ ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
//@ diverges true;
public static void setto1(int[] a){
   int    i = 0;
   while  (i < a.length) {
     a[i] = 1;
    i++;
    }
}
```

Esercizi

Calcolo del quadrato tramite somme successive

```
//@ requires n >= 0;
//@ ensures \result == n * n;
public static int quadrato(int n){
  int i = 0;
  int result = 0;
  while (i != n){
    result += n;
    i++;
  }
return result;
}
```

Esempio: divisione intera

```
//@ public normal_behavior
//@ requires dividend >= 0;
//@ requires divisor > 0;
//@ ensures (\exists int resto; resto >= 0 && resto < divisor; dividend ==
  \result * divisor + resto);
public static int divide(int dividend, int divisor) {
  int quoziente = 0;
  int b = dividend; // resto
  while(b>=divisor){
     b = b-divisor;
     quoziente++;
  }
return quoziente;
}
```

Esempio: divisione intera

```
//@ public normal_behavior
//@ requires dividend >= 0;
//@ requires divisor > 0;
//@ ensures (\exists int resto; resto >= 0 && resto < divisor; dividend ==
  \result * divisor + resto);
public static int divide(int dividend, int divisor) {
  int quoziente = 0;
  int b = dividend; // resto
/*@ loop_invariant
  @ (dividend == quoziente * divisor + b) && b >= 0;
  @ decreases b;
  @*/
  while(b>=divisor){
     b = b-divisor;
     quoziente++;
  }
return quoziente;
}
```

Come “intuire” l'invariante

- L'invariante deve implicare all'uscita la postcondizione
  - Domandati perchè alla fine la post condizione vale?
- Prova a percorrere il ciclo con qualche caso di test, con un “giro” del ciclo, con due e così via

Come trovare invarianti

Using the experience from the previous examples we can sketch on a general plan for how to prove a loop correct:

- Look at the postcondition to see how it could be generalized.
- Perhaps dry-run the loop a few times to see a pattern of what is being preserved.
- From this choose a first attempt of the loop invariant.
- Try to prove that it's initially valid, preserved and entails the postcondition.
- While checking invariant preserved and use invariant you might encounter necessary extra conditions which you add to the invariant.
- Make sure that the new conditions are also initially valid and preserved.
- Repeat this until the whole proof goes through.

Altri esercizi

- assegna

```
{x>=0}
y:= 0;
while (y < x) {
   y:= y+1;
}
{y=x}
```

doppio

```
int old_x = x;
int dop = x;
while (x != 0) {
dop := dop +1 ; x := x-1;
}
{dop = 2 *old_x}
```

Esercizio 2

Multiplication

```
{y >=  0}

  z = 0;
  n = y;
  while (n>0) {
    z = z + x;
    n = n-1;
  }

{z = x * y}
```

Integer division

```
{x >= 0 & y >= 0}

  a = 0;
  b = x;
  while (b >= y) {
   b = b - y;
   a = a + 1;
  }

{x = a * y + b & b >= 0 & b < y}
```

Array

- Nel caso di array, considera quello che continua a valere a seconda dell'indice.
- Esempi

```
//@ public normal_behavior
  //@ requires a !=null;
  //@ ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  //@ diverges true;
public static void setto1(int[] a) {


//@ public normal_behavior
  //@ requires a !=null;
  //@ requires a.length >0;
  //@ ensures (\forall int x; 0<=x && x<a.length; \result >= a[x]);
  //@ ensures (\exists int x; 0<=x && x<a.length; \result == a[x]);
  //@ diverges true;
public static int getMax(int[] a) {
```

Array + Classi

- Spesso usiamo array nelle classi
- Esempi vedi temi d'esame

Soundness - Consistenza

la logica di Hoare è consistente (sound) nel senso che ogni cosa che può essere provata è corretta (cioè è vera)

Dim.: ogni assioma conserva la verità delle asserzioni (consistente).

Completezza

Un sistema di dimostrazioni è detto completo se ogni asserzioni vera può essere dimostrata

- la logica proposizionale è completa
- Per l'aritmetica invece non c'è sistema di regole che sia completo (Godel).

- puoi leggere “Gödel, Escher, Bach: An Eternal Golden Braid”, in italiano: Godel, Escher, Bach un’Eterna Ghirlanda Brillante di Douglas R. Hofstadter http://en.wikipedia.org/wiki/G%C3%B6del,_Escher,_Bach

E la logica di Hoare?

Due problemi:

1) incompletezza del calcolo aritmetico

2) problema della terminazione:

{p} S {false} significa che S non termina a partire dalle precondizioni p.
Questo è indecidibile, cioè le regole non riescono a provarlo sempre:

il sistema però è relativamente completo

Total correctness con KeyJML

- correttezza totale
  - Oltre ad essere corretto termina anche
- Esempio di corr. Parziale ma non totale:

```
\ensures n == 0;
while(n != 0) {n--}
```

- Se voglio provare la correttezza totale devo dire perchè il ciclo si ferma
  - decreases ...