Module 14 - Automatic Program Verification using Hoare Logic

# Overview and Objectives

## OVERVIEW

In this module we will learn about automatic verification, which is a practical and real-world use of the specifications and reasonings — topics that we have learned throughout the class. We will start with the basic of logical reasoning and then learn how to use verify program correctness with respect to given specifications and contracts).

## COURSE LEVEL OBJECTIVES (CLO)

Upon completion of this course, you should be able to:

1. Construct modern high quality software systems and reason about them.
2. Properly define software specifications and rep-invariants.
3. Leverage immutability to properly construct threat safe programs.
4. Explain object-oriented concepts such as information hiding, encapsulation, data and type abstraction, and polymorphism.
5. Properly use exception handling
6. Identify when it is appropriate to use inheritance and generics.

## MODULE LEVEL OBJECTIVES (MLO)

Upon completion of this module’s activities, you should be able to:

1. explain and demonstrate basic logic terminologies and usage (CLO 1,2)
2. compare and describe key concepts and terminologies in automatic program verification (e.g., loop invariants, partial vs total correctness) (CLO 1,2)
3. Transform programs and specifications to logical formulae (CLO 1,2)
4. Construct requirements and setups for verficiation and verify program automatically using Hoare logic (CLO 1,2)

# Module Video (Wiley-Produced w/Dan Ramos) [3-5 minutes]

# Learning Materials [~100 pages, ~3.5 hours]

## TEXTBOOK READINGS

* Hoare logic notes: <https://nguyenthanhvuh.github.io/posts/program-analysis-notes.html#hoare>
* Barbara Liskov with John Guttag. Program Development in Java. Addison Wesley, 2001, ISBN 0-201-65768-6.
  + Chapter TODO

## DONE ADDITIONAL RESOURCES

* Propositional logic: <https://en.wikipedia.org/wiki/Propositional_calculus>
* Hoare logic: <https://en.wikipedia.org/wiki/Hoare_logic>

# Introduction (MLO 1) [~15 mins]

* Previously we have look at program specifications that involve pre and post conditions and **manually** reason that the program implementation is correct (adheres to) with respect to the specification.
* As we also have experienced, such manual reasoning is difficult, time consuming, and error-prone, even for short "toy" examples shown class.
* Automatic testing is popular and widely-used, however not adequate for many domains and situations such as avionic systems and medical systems
* Automatic verification is what required, and has enjoyed many successes in the real-world. For examples,
  + Hardware (e.g., circuit designs, CPU) are rigorous verified by chip makers
  + Companies and government organizations develop and rely on verification tools (e.g., Mars Rover, Avionic system of Airbus, Facebook newsfeeds and instagrams, Microsoft drivers are all verified in some ways).
* Today we will learn about Hoare Logic, one of the most popular ways to \*automatically verify program correctness.
* Hoare logic:
  + a formal systems with a set of rules to formally reason about the correctness of a program with respect to given specifications
  + invited by Sir. Tony Hoare, who is the Turing award winner in 1980 , and in addition to Hoare logic, has developed many well-known concepts such as the \*quicksort algorithm and "null pointer" (the billion dollar mistake)
  + standard for many practical verification techniques and tools

For this topic, you will learn about

* Hoare tripple: which describes how the program changes its states during execution
* Total vs. Partial correctness: Total correctness reasons both about program termination (e.g., the loop will eventually terminate) and that the program is correct and satisfies the postcondition when it terminates; whereas parial correctness \*assumes the program terminates and only deals with program correctness.
* Computing Weakenst Preconditions using Hoare rules: a set of rules for each types of statements in a program that leads to the weakest preconditions required for verification
* Finding and using Loop invariants: the crucial piece of information that the user needs to provide to enable verification automation under Hoare logic
* Formulating specifications, weakest preconditions, loop invariants as logical formulae
* Using SAT solver to automatically prove program correctness

# Intro to Logic (MLO 2) [~1 hour]

## Terminology

* variables:
  + boolean variables: can take either True (1) or False (0) value
  + integer variables: can take integers
  + …
* Logical connectors: and (&), or (|), not (!), imply (=>)
* Formulae: proper combination of variables and logical connectors.
* x  
  x & !x  
  (x | y) & z  
  x => y  
    
  x > 6 and x < 5 : False (no value of x would satisfy this)  
  x > 6 => x > 1 : True (every value of x would satisfy this)  
  x > 6 and y = 3 : False (counterxample: {x=5 , y=2})  
  x > 6 => y = 3 : False (counterexample : {x=7, y=4})
* Satisfiable, Valid (Tautology), Falsification
  + a formula f is \*satisfiable if there is \*some assignment to the values in f that makes f evaluate to True
  + - x <= 6 or y = 3 is satisfiable (e.g., x=4, y=4)  
    - x > 6 and y = 3 is SAT (e.g., x =7, y=3)   
    - x > 6 => x > 1 is SAT (e.g., x=7)  
    - x > 6 and x < 5 is UNSAT
  + a formula f is \*valid if f is always satisfiable for \*every assignment
  + x > 6 => x > 1 is valid  
    x = x is valid
  + a formula f is a \*falsification if f is always unsatisfiable for \*every assignment
  + x > 6 and x < 5  
    x != x  
    x = x + 1 (if x = some infinite number, then x = x + 1 would True, so assume x is finite)

## Important Concept: Formula evaluation

* Understand how logical connector works, especially **implication**
* Understand well how to evaluate formula through the above definitions of satisfiable, valid, falsification.

## Instructor Screencast: TITLE

Link to MP4 File

## Interactive Element: TITLE

## Important Concept: Text

Text.

## Instructor Screencast: TITLE

Link to MP4 File

## Important Concept: Text

Text.

# Learning Unit 2 – Program Verification using Hoare logic (MLO 2, 3, 4) [~3 hours]

## Terminology

### Hoare Tripple {P} S {Q}

### Partial and Total Correctness

* **Partial**: assume P holds and S *successfully* executes, Q holds
  + Here, we \*assume P and the program terminates (S successfully executed)
* **Total**: assume P holds, if S *successfully* executes, then Q holds
  + Here, we \*require S terminates
  + Dififcult because having to ensure the termination of S

### Examples of Hoare Tripples

* Consider a program S with a single assignment statement x:=5. The Hoare tripple {True} x := 5 {x > 6} is *not* a valid tripple, but these next ones are
* {True} x := 5 {x=5 or x= 6 or x > 6}  
  {True} x := 5 {x > 1}  
  {True} x := 5 {x = 5}
* Moreover, the postcondition in x=5 is \*strongest because it is more precise than x > 1 and (x=5 or x=6 or x > 6). In general we want the strongest (most precise) postcondition.
* Consider another program z:= x/y These are valid Hoare tripples:
* {x = 1 & y = 2} z:= x/y {z < 1}  
  {x = 2 & y = 4} z:= x/y {z <1}  
  {0 < x < y & y != 0} z:= x/y {z <1}
* Moreover, the precondition 0 < x < y & y != 0 is the \*weakest precondition (i.e., it is the least constraint precondition). In general we want the weakest precondition.
* These are invalid tripples:
* 1. {x < y} z:= x/y {z < 1} (counterexample input x=-1, y=0, after executing z:=x/y, we do not have z < 1 and instead got a div-by-0 exception)  
  {x = 0} z:= x/y {z < 1} (counterexample input x=0, y=0)  
  {y != 0} z:= x/y {z < 1} (counterexample input x=2 , y=1)  
  {x < y & y != 0} z:= x/y {z <1} (counterexample input x=-2, y=-1)

## Important Concept: Text

## Rules to Computing Weakest Preconditions

We can automatically verify (partial) program correctness using Hoare Triples and weakest preconditions. To prove {P} S {Q} is valid, i.e., to prove the program S is correct wrt to the precondition P and postcondition Q, we check that P => wp(S, Q) is valid. Here, the function \*wp returns the weakest precondition allowing the program S to achieve the postcondition Q.

Thus, to show the validity of {P} S {Q}, we show that P implies (=>) the WP of S wrt to Q. Hoare defines rules to obtain the WP of different kind of (imperative) program statements as shown below.

### Quick summary of Hoare rules

|  |  |  |  |
| --- | --- | --- | --- |
| Statement | S | wp(S, Q) | Comment |
| *Assignment* | x := E | Q[x/E] | replace all occurences of the variable x in Q with the expresion E |
| [List of Assignments](#list) | S1;S2 | wp(S1, wp(S2,Q)) |  |
| *Conditional* | if b then S1 else S2 | b => wp(S1,Q) & !b => wp(S2,Q) |  |
| *Loop* | while b do S | (I) & (I & B => wp(S,I)) & (I & !B => Q) | I is a user supplied loop invariant |

### ASSIGNMENT

An assignment statement x := E is one of the most popular types of statement. It assigns the value of an expression E to a variable x. The WP for an assignment wp(x:=E,Q) is obtained by substituting all occurences of x in Q with the expression E.

WP(x := E, Q) = Q[x/E]

Example:

WP(x:=3, x + y = 10)  
= 3 + y = 10   
= y = 7  
Thus, we have {y=7} x := 3 {x + y = 10}

WP(x:=3, {x + y > 0)   
= 3 + y > 0   
= y > -3  
Thus, we have {y > -3} x := 3 {x + y> 0}

### LIST of statements

A list of sequential statements. The WP for list is defined *recursively* as follows.

WP([S1; S2; S3 ...;] Q) = WP(S1, WP(S2;S3;.., Q))  
WP([], Q) = Q

Example:

WP([x:=x+1; y:=y\*x], y=2\*z)   
= WP(x:=x+1, WP([y:=y\*x], y=2\*z))  
= WP(x:=x+1, y\*x=2\*z)  
= y\*(x+1)=2\*z  
Thus, we have {y\*(x+1)=2\*z} x:=x+1; y:=y\*x {y=2\*z}

### CONDITIONAL

The WP of a conditional statement if b then S1 else S2, Q combines the WPs of S1 and S2.

WP(if b then S1 else S2, Q) = (b => WP(S1,Q)) & (!b => WP(S2, Q))

Example:

WP(if x > 0 then y := x + 2 else y := y + 1, y > x)   
= (x>0 => WP(y:=y+x, y>x) & (x<=0 => WP(y:=y+1, y>x))  
= (x>0 => y+x>x) & (x <= 0 => y+1>x)  
= x>0 => y>0 & x<=0 => y+1>x

WP(if x > 0 then y :=x else y:= 0, y > 0)   
= (x>0 => WP(y:=x, y >0)) & (x<=0 => WP(y:=0, y>0))  
= (x > 0 => x > 0) & (x <= 0 => 0 > 0)  
= True & x > 0   
= x > 0 # (0> 0 is false, and so !(x<=0) or false is !(x<=0) = x>0)

Note: Instead of using => (imply), which might be confusing to some, we can use just ! (not) and || (or)

WP(if b then S1 else S2, Q)  
= (b => WP(S1,Q)) & (!b => WP(S2, Q))  
= !((b & !WP(S1,Q)) || (!b & !WP(S2, Q)))

## Important Concept: Loop Invariants

### Loop Invariants

At a high level, loop invariant capture the meaning of the loop, and thus help understand and reason about the loop. They are especially helpful for automatic verification.

A loop invariant is a property I that always holds at the **loop entrance**. This means that I (i) holds when the loop entered and (ii) is preserved after the loop body is executed (i.e., I is an inductive loop invariant).

### Where is the loop invariant I?

If you have a loop that looks like

while (b){  
 // I   
 //loop body  
}

It is useful to transform it to this equivalent form

while (True){  
 // [I] : loop invariant I is right here  
 if (!b) break  
 //loop body  
}

Then the loop invariant I is right when you enter the loop, as indicated by [I] in the code above.

Note that I is not located \*after the guard condition b is satisfied, e.g.,

while (b){  
 //[I] : incorrect location for loop invariant  
 //loop body  
}

### What is the loop invariant I?

We will use an example to demonstrate loop invariants. Consider a simple program

// {N >= 0}   
i := 0;  
while(i < N){  
 i := N;  
}  
// {i = N} // post condition

To make it easier to see where loop invariants are, we first transform this program into an equivalent one:

// {N >= 0}   
i := 0;  
while(true){  
 // [I]: loop invariants here  
 if(!i < N) break;  
 i := N;  
}  
// {i = N} // post condition

The while loop in this program has many possible loop invariants (any property that is true at [I]):

1. true : is always a loop invariant, but it is very weak and trivial, i.e., almost useless for any analysis
2. N >= 0: because N>0 is a precondition and N is never changed
3. i>=0: because i is initalized to 0 can only changed to N, which itself is >=0 and never changed.
4. i <= N: because i can only either be 0 or N, and N >=0.

### Which loop invariants to use?

An important question to ask is which of these invariants are useful? Typically, the more stronger the better as they capture the meaning of the loop more precisely (thus true is not very useful). However, the answer really depends on the task we are trying to achieve. If the task is to prove a very weak property, then we might not need strong loop invariants, e.g., for instance to prove that N >= 0 as the postcondition, then we only need the loop invariant N >= 0. Vice versa, if the task is to prove a strong property such as i=N, then we likely need strong loop invariants, e.g., i<=N.

In many cases, we can guess which loop invariants are useful based on the postconditions we want to prove. However, in the general cases we do not know a priori which loop invariants to use. If the program is indeed correct wrt the specs (i.e., the representing Hoare tripple is valid), there are two possible scenarios about using loop invariants to prove programs:

1. if we use sufficiently strong loop invariants, then we will be able to prove the program is correct.
2. if we use insufficiently strong loop invariants, then we will *not* be able to prove the program is correct.

The *loop* section in Hoare logic gives concrete examples demonstrating these two cases.

Thus, this gives an \*crucial observation: if we can prove that a program is correct (e.g., using Hoare logic), then it is really correct. However, if we cannot prove that the program is correct, then we do not know whether the program is correct or not (it could really be wrong, or it is actually correct but we can't prove it because we use rather weak loop invariants).

### Important Concept: Computing WP for LOOPs

1. LOOP

* Unlike other statements where we have rules to compute WP automatically, for loop, we (the user) need to supply a *loop invariant* ~I~to obtain the WP of loop. This *subsection* describes loop invariants. The WP for loop is:
* WP(while [I] b do S, Q) = I & (I & b => WP(S,I) & (I & !b) => Q)
* As can be seen, the WP for loop consists of 3 conjuncts:
  1. I : the loop invariant (should hold when entering the loop)
  2. (I & b) => I : (entering the loop because b is true) I is preserved after each loop body execution
  3. (I & !b) => Q (exiting the loop because b is false), when exiting the loop, the post condition holds
* Thus, to compute WP for loop, you would need to come up with invariants. Moreover, as *mentioned*, we will need to pick a sufficiently strong loop invariants to be able to prove the program. Note that we will always able to create the weakest WP, but it might not be good enough to prove the program at the end.
* Below we demonstrate the computation of WPs using sufficiently and insufficiently strong invariants. We use the same example program used [here](#li):
* // {N >= 0}   
  i := 0;  
  while(true){  
   // [I]: loop invariants here  
   if(!i < N) break;  
   i := N;  
  }  
  // {i = N} // post condition
* This program has several loop invariants at [I] including N >= 0, i>=0, i <= N. Also, the program can be written as S: i := 0; while[i<=N] i < N do i:= N], with precondition P: N >= 0 and postcondition Q: i==N.

1. Example: using a *sufficiently* strong invariant

* Here, we use the loop invariant i <= N to prove S is correct wrt to P,Q. As we will see, this loop invariant is sufficiently strong because it allows us to prove the program.
  1. Apply the WP to the program, which is a list of statements.
  + WP([i := 0; while[i<=N] i < N do i:= N], i = N)   
    = WP(i := 0; WP(while[i<=N] i < N do i:=N], i = N) //WP rule for list of statements
  1. Apply the WP to while
  + // Let's first compute WP(while[i<=N] i < N do i:=N, {i = N}). According to the WP rule for LOOP, we will have 3 conjuncts   
    1. i <= N  
      
    2. (i <= N & i < N) => WP(i:=N, {i<=N})  
     = i < N => N <= N   
     = i < N => True   
     = True // because !(i<N) or True is true (anything or with true is true)  
      
    3. (i <= N & !(i<N)) => i = N  
     = i = N => i = N  
     = True // because !(i=N) | i = n is True (a or !a is True)  
      
    = i <= N & True & True  
    = i <= N
  1. After obtaining the WP i<=N for while, we continue with WP(i:=0, i<=N)
  + // WP([i := 0; while[i<=N] i < N do i:= N], i = N) = WP(i := 0, i<=N)  
    WP(i := 0, i<=N)  
    = 0<=N //WP rule for assignment
  1. Now we construct a *verification condition* (vc) to check that the given precondition P: N >= 0 implies the WP 0<=N
  + P => WP([i := 0; while[i<=N] i < N do i:= N], {i = N})   
    = N>=0 => 0<=N // N>=0 is the given precondition and 0 <= N is the WP we obtain above for the program  
    = True
  + Because te given precondition N>=0 implies 0<=N, the Hoare tripple is valid, i.e., the program is correct.
  1. Also, the loop invariant i <= N is thus \*sufficiently strong to let us prove the program satisfy the specifications.

1. Example 2: using an *insufficiently* strong invariant

* Here, we use the loop invariant N >= 0 to prove program. As we will see, this loop invariant is not sufficiently strong because we will not be able to use it to prove the program.
  1. Apply the WP to while
  + WP(while[N >= 0] i < N do i:=N, {i = N})  
    =  
     1. N >= 0  
     2. (N >=0 & i < N) => WP(i := N, N >= 0) =   
     (N >=0 & i < N) => i >= 0 // we can't simplify much, so just leave as is  
      
     3. N >=0 & !(i<N) => i =N  
     (N >= 0 & i >= N) => i = N  
     i>= 0 => i = N // we can't simplify much, so just leave as is  
      
     = N >=0 & (N >=0 & i < N) => i >= 0 & (i>= 0 => i = N)  
      
    WP(i:=0; {N >=0 & (N >=0 & i < N) => i >= 0 & (i>= 0 => i = N)})  
     = (0 >= 0) & (0 >= 0 & 0 < N => 0 >= 0) & (0>=0 => 0 = N) //apply WP for assignment and simplify  
     = TRUE & TRUE & 0 = N  
     = 0 = N
  1. Obtain the vc
  + P => 0 = N // the given precondition implies 0 = N  
    (N >= 0) => 0 = N // This is not valid, e.g., counterexample N=3
  + The vc is not valid and thus we were not able to prove the Hoare triple and hence do not know whether the program is correct or not.
  1. Thus this loop invariant is not sufficiently strong for us to prove the program.
  + **Important**: as mentioned [here](#li-to-use), not being able to prove simply means we cannot prove it using this loop invariant. It \*does not mean that you disprove it or show that the Hoare triple is invalid. (in fact, we know the Hoare tripple is valid if we used a different loop invariant, e.g., i <= N )

### Instructor Screencast: TITLE

Link to MP4 File

### Interactive Element: TITLE

### Important Concept: Text

Text.

### Instructor Screencast: TITLE

Link to MP4 File

# Assignment – Program Verification using Hoare Logic (MLO 2, 3, 4) [~2.5 hours]

## Purpose

The purpose of this assignment is to help you practice learned concepts on logical formula, Hoare tripple, and especially program verification using Hoare logic.

## Instructions

### Q1

Fill in P,S,Q to make the following Hoare tripples valid.

1. {Q} x:=3 {x = 8}
2. {P} x:= y - 3 {x = 8}
3. {x = y} S {x = y}
4. {x < 0} while(x!=0) do x := x - 1 {Q}

### Q2

Consider the program:

// {N >= 0} # P  
i = 0;  
while (i < N){  
 i = i + 1;  
}  
  
//{i == N} # Q

* Identify the loop invariants for the loop in this program
* Use a sufficiently strong invariant to prove the program is correct
* Attemp to prove the program using an insufficiently strong invariant, describe what happens and why.

## Deliverable

Submit a short essay (2 pages maximum) with your response to questions above

## Due Date

Your assignment is due by Sunday 11:59 PM, ET.

# Module 1 Quiz (MLO 2, 3, 4) [~.5 hour]

## Purpose

Quizzes in this course give you an opportunity to demonstrate your knowledge of the subject material.

## Instructions

Note the following instructions for your quiz: The quiz is 20 minutes in length. The quiz is closed-book.

* Fill in the ??? and briefly explain your answer

{???} y := x + 1 {y=43}  
{x + 1 <= N} ??? {x <= N}

* Is the following a valid Hoar tripple ? explain

{0 <= x <= 15} if (x<15) then x:=x+1 else x:=0 {0 <= x <= 15}

* For the below loop, give 2 non-trivial (so no True) loop invariants, one of which should involve a relationship of both i and j. Hint: convert the for loop into a while loop first.

int j = 9;  
for(int i=0; i<10; i++)   
 j--;

## Deliverable

Use the link above to take the quiz.

## Due Date

Your quiz submission is due by Sunday 11:59 PM, ET.