Ser 515 Course Notes Axiomatic Specifications

Last Modified October 17, 2016 4:00 pm

6. Software Formal Descriptions
6.a  Formal Basis - Axiomatic Descriptions

6.a.1 References

- Anthony Aaby, “Theory Introduction to Programming Languages”, draft version 0.9, Edited July 15, 2004. See chapter 3. Available at: http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/Reading%20Materials/book%5B1%5D.pdf


6.a.2 Motivation for Software Engineers to know Formalisms

- A view of Software Engineering for critical systems indicates that a formal specification of a software system should drive its construction. Construction should be (partly) automated and driven by formal models of the system.

- Formalisms are used as the basis for many aspects of software engineering:
  - Formal requirements specifications,
  - Many approaches to testing, reliability and certification are based on various formalisms, for example structural and domain testing,
  - Model-Driven Software Engineering has formal descriptions of software as its basis,
  - Cost estimation and complexity estimation models,
6.a.3 Classical Approaches to Language Semantics

- Attributed and Translational Grammar Approach,
- Operational (Virtual Machine),
- Denotational Semantics,
- Axiomatic.
Attributed and Translational Grammar Approach

- Grammars for describing language syntax
  - Grammars are made up of Productions,
  - Productions describe legal syntactic structure of sentences (programs) in the language:
    
    sentence --> direct_object verb subject
  
  - This production is read as: “A sentence is defined to be a direct_object followed by a verb, followed by a subject.”
  
  - Sentence is a syntactic category, as are direct_object, verb and subject. These may be defined elsewhere in the grammar, for example:
    
    verb --> RUN | WALK | HIT | SING

- Grammar based semantics provide semantics in terms of the math functions that augment Productions to provide the meaning of the underlying syntax:
  
Operational (Virtual Machine)

- Uses the Interpreter architectural style discussed in class.
- We define the meaning of a program (or any software system) in terms of operations on some abstract (virtual) machine.
- In effect, we write a program using the operations of the virtual machine whose execution reflects the meaning of the language statement.
- Virtual machine may not have a realized implementation.
- Virtual machine must be defined in sufficient detail (partly in terms of the operations it can execute) to provide clear semantics of any program written using the operations.
Denotational Semantics

- The meaning of statements (programs) in the language are described by mathematical functions to express the semantics.

- For example,
  \[ C[I := E] \ u \ s = d \subseteq L \text{ and } e \subseteq R \rightarrow s[d \mapsto e], \text{ error} \]
  where \[ d = u[I] \]
  and \[ e = E [[E]] \ u \ s \]
6.a.4 Axiomatic Semantics

- Define the meaning in terms of a **proof system** (a **theory** or **math logic system**)
- Use predicates (conditional expressions or first order logic expressions) to define the meaning of statements (programs):
  \[ P_1 \{ x := E \} P_2 \]
  if \( P_1 \) is true before execution of the statement \( x := E \) then \( P_2 \) will be true when (and if) the statement completes.
- Some authors reverse the use of curly braces in this notation:
  \[ \{P_1\} x := E \{P_2\} \]
What is an Axiom System (a math logic)?

- Three constituents of a logic system:
  - A language for expressing theorems. This language is the set of all statements of the form:
    \[ P \{ S \} Q \]
    where:
    - \( P, Q \) are predicates whose free variables are those of the program and \( S \) is the programming language statement(s).
  - A set of Axioms - Statements in the language of theorems that express the assumed truths of the system
  - Rules of Inference - rules that allow us to deduce new findings from established findings.
What is an Axiom System (a math logic)?

- **Proof**: A proof is an ordered sequence statements in the language of **Theorems**. The ordered sequence is a derivation in which each step is either an **axiom** or follows by applying a **rule of inference** to previous statements in the sequence.

- **Theorem**: A statement in the **language of theorems** for which there is a proof (having the theorem as the last statement of the sequence). The notion of theorem-hood in an axiomatic system is mindless (mechanical) in the sense that we could write a program that enumerates all theorems of a theory.

- The problem of determining whether a **statement** in the language of an axiom system is a **theorem** is a difficult problem.

- For example a **proof** of **Tn** may take the form: 
  
  \(( T1, T2, T3, ..., Tn )\) where each **Ti** is of the form **P \{S\} Q**

- When there exists a proof for **Tn**, we write
  
  \(|-- Tn\)

  (\|-- is a turnstile symbol) to indicate **theorem-hood**.
What is an Axiom System (a math logic)?

- **Validity**: The notion of validity for an axiom system hinges on what we want theorems to mean.

- For example, with Hoare logic we want:
  
  \[
  \vdash P \{S\} Q
  \]

  To mean “if $P$ is true before $S$ begins execution then $Q$ will be true when (and if) $S$ completes.

- An axiom system is said to be **sound** when all theorems of the system are **valid**.

- Although not quite as straightforward, you can consider that an axiom system is said to be **complete** when everything that is **valid** is a **theorem** of the **theory**.
Hoare Logic - Significance to Software Engineering

- Hoare developed an axiom system for computer programs in a Pascal-like language.

- What is the significance for software engineering? Consider the form of a theorem in Hoare Logic:
  \[ \vdash P \{S\} Q \]

- \( P \) and \( Q \) provide the input and output predicates that describe the functionality of the statement(s) \( S \). Thus, if \( S \) is the software system (program(s)) then \( P \) and \( Q \) provide the functional requirements.

- \( P \) and \( Q \) are formal representations of what normally appears in the SRS (and SDS). \( \{S\} \) is the code that gets developed during the coding phase.

- Demonstrating the theorem-hood of \( P \{S\} Q \), and even expressing \( P \) and \( Q \) formally for complex software systems is difficult. But, this provides much of the theory and practice behind testing and requirements specification.
Hoare Logic: A Theory for Computer Programs

- Axioms for Integer arithmetic:
  - Commutes: |-- X + Y = Y + X and |-- X * Y = Y * X
  - Associative: |-- (X + Y) + Z = X + (Y + Z)
  - Distributive: |-- X * (Y + Z) = X * Y + X * Z

- There are several other axioms of similar form making concrete the properties of the predefined types and operations on those types. Note that this information is specific to the language and its definition (or at least should be specific to the language definition and not the language implementation).

- Possible axioms to describe integer arithmetic overflow:
  - No overflow: |-- there does not exist X such that X = MAX + 1
  - Finite boundary: |-- MAX + 1 = MAX
  - Modulo: |-- MAX + 1 = 0
Hoare Logic: Sample Rules for changing specifications

- **Stronger Antecedents (SA)**
  - if |-- P \{S\} Q and |-- T => P (\(\Rightarrow\) denotes logical implication T implies P) then |-- T \{S\} Q
  - The *antecedent* is the *precondition* (the predicate describing what must be true before executing \(S\)). A stronger predicate (\(T\) is stronger than \(P\) in \(T \Rightarrow P\)) is one that is more constraining, has more properties, or describes a smaller subset of a finite state space. This rule tells us \(S\) can be used with any *precondition* that is *stronger* than \(P\) to obtain the same results.

- **Weaker Consequents (WC)**
  - if |-- P \{S\} R and |-- R => Q then |-- P \{S\} Q
  - The *consequent* is the *postcondition* - what must be true after \(S\). This rule tells us that \(S\), with the same *precondition*, will satisfy any *postcondition* that is less constraining than \(Q\).
Hoare Logic: A Rule for composing statements

- **Sequential Composition (;)**
  - if |-- P {S1} T and |-- T {S2} Q then |-- P { S1 ; S2 } Q

- This rule tells us how to glue two statements (programs) together to be executed in sequence - and the relationship of doing so to their corresponding requirements.

- **if-the-else (ITE)**
  - given: if B then S1 else S2
    - if |-- P and B {S1} Q and |-- P and not B {S2} Q then |-- P {if B then S1 else S2} Q
Hoare Logic: Axiom schema for Assignment

- **Assignment** is handled in Hoare logic by an *axiom*. These is a single form for an assignment that is applied to any assignment statement.

- Assignment Axioms have the form:
  \[ |\rightarrow P0 \{ x := f \} P \]
  where: \( P0 \) is obtained from \( P \) by substituting \( f \) for all occurrences of \( x \).
  Sometimes, \( P0 \) is written as \( P(f/x) \) (\( P \) with \( f \) substituted for \( x \))

- For Example,
  \[ |\rightarrow 5 > 25 \{ I := 5 \} I > 25 \]
  this says you can not start \( I:=5 \) in a state that will result in \( I>25 \)
  \( 5>25 \) is the same as **false** -- no states satisfy the predicate false.

- Another Example,
  \[ |\rightarrow ( I > 20 ) \{ I:= I + 5 \} ( I > 25 ) \]
  Note \( ( I > 20 ) \) is the same as \( ( I + 5 > 25 ) \)
Hoare Logic: Rule for Iteration

- **While-do (wd)**
  - given: `while B do S`
  - if `|-- P and B {S} P`
    then `|-- P {while B do S} P and not B`
  - `P` is the **invariant assertion**. `P` is true each time you enter the loop (before executing any statements).

- This rule reflects the **inductive** nature of the **while-do**, or any **iteration**. The rule actually forces the use of an inductive argument to discuss the meaning of a **while-do**.
Hoare Logic: Rule for Iteration

- What is **induction**? Suppose you want to show a proposition $A$ is true for all natural numbers $0$, $1$, $2$, ... That is, $A_0$, $A_1$, $A_2$, ...
  This is done by showing the two following:
    - **Basis step** - show $A_0$ is true
    - **Inductive step** - show:
      assuming $A_n$ is true, argue to show $A_{n+1}$ must be true.

- The while-do rule provides the inductive step.
Hoare Logic: Rule for Iteration

• For example, consider the code segment below which finds the maximum value in a 10 element array:

\[ I = 1; \]
\[ \text{max} = A(1); \]
\[ \text{while } (I < 10) \{ \]
\[ \quad I = I + 1; \]
\[ \quad \text{if } A(I) \geq \text{max} \text{ then } \{ \]
\[ \quad \quad \text{max} = A(I); \]
\[ \quad \} \]
\[ \} \]

• The **post condition** for this code sequence should be:

\[ \text{max is the largest value in the array } A(1..10) \]
Hoare Logic: Rule for Iteration

• What is the invariant assertion (call it P) for this loop?
  \textbf{max is the largest in the subarray A(1..I) and I <= 10}

- Note: The \textit{invariant P} is true just before entering the while,
  The \textit{invariant P} and \textbf{I >= 10} (not B) implies the \textbf{post condition}
  The while rule (wd) requires that we perform the inductive step
  The basis step is showing \textbf{P} after \textbf{I=1; max=A(1)}
Example Using a Toy Program:

- Toy Program S1 has 2 initial assignments followed by an if. The then clause is always executed and one result is I = 5

S1: I := 5;
    J := I * 2;
    if J > 9  then I := I + 1;
              else J := J + 1;

- Hoare Logic proofs can show correspondence between specifications and code. More commonly we use it for testing.

- Specifications for the program may be: Precondition = True, Postcondition = I is 5. In Hoare Logic, we’d express this as

  \[\text{true } \{S1\} \ I=5\]

- Although its not the recommended use of Hoare Logic, below is a proof using the system. Commonly, predicates are used to describe requirements, constraints. Proofs are sometimes used for certifying
Hoare Logic: Step-by-Step

- Demonstrate \textbf{true \{S1\} I=5} using Hoare Logic

First show Lemma A: I=5 ^ J=10 \{if-then-else\} I=6

1 \|-- I=5 \{I:=I+1\} I=6 (by axiom :=)
2 \|-- ( I=5 ^ J=10 ^ J>9 ) => I=5 (by predicate calculus (pc))
3 \|-- I=5 ^ J=10 ^ J>9 \{ I:=I+1 \} I=6 (by SA on 1,2)
4 \|-- for any P, false \{ J:=J+1 \} P (by SA and ax :=)
   note, false => P for any P.
5 \|-- ( ( I=5 ^ J=10 ^ J<=9 ) => false ) (by pc)
6 \|-- ( I=5 ^ J=10 ^ J<=9 ) \{ J:=J+1 \} I=6 (by SA on 4,5 - P is I=6)

Lemma A: \|-- I=5 ^ J=10 \{if-then-else\} I=6 (by ITE on 3,6)

7 \|-- ( I=5 ^ I*2 =10 ) \{ J:=I*2 \} ( I=5 ^ J=10 ) (by ax :=)
8 \|-- ( 5=5 ^ 5*2=10 ) \{ I := 5 \} ( I=5 ^ I*2 =10 ) (by ax :=)
9 \|-- true => ( 5=5 ^ 5*2=10 ) (by pc)
10\|-- true \{ I:= 5 \} ( I=5 ^ I*2 =10 ) (by SA on 8,9)
11\|-- true \{ I:= 5; J:=I*2 \} ( I=5 ^ J=10 ) (by ; on 10, 7)
12\|-- true \{ I:= 5; J:=I*2; if-then-else \} I=6 (by ; on 11, Lemma A)
Approaches to Describing Methods

- **Abstraction** - the functionality of a procedure/method/function is described in terms of two predicates, a *precondition* and a *postcondition*.
  - The *precondition* describes constraints on the inputs to the procedure (in terms of input parameters and global variables).
  - The *postcondition* describes the results of the procedure in terms of changes to global variables and output parameters.

- **Predicate Transformer** - An axiom schema is developed, which describes the effects of the commands of a procedure’s implementation. To illustrate, for assignment we use the schema:

  \[ \text{|-- } P(f/x) \ { x = f } \ P \]

  to describe the effect of the assignment. The effect is described by showing the transformation to the postcondition to obtain the precondition. In the same manner, a transformer for a procedure is developed.
Approaches to Describing Methods

- The difference between abstraction and predicate transformer can be described in terms of the meaning of the statement: $X = 20 \times 3 + 15$.
- Abstraction says the meaning is 75. That is the meaning is the new value of the variable $X$.
- Predicate transformer says the meaning is the result of tripling 20 and adding 15 and associating the resulting value with $X$.
- While the predicate transformer approach may be more precise, abstraction has the value of summarization.
Abstraction

• Consider the method:

```cpp
func P (int L, out int M, inout int N) {
    int x;
    pre (L>0 and N>0);
    X = N + L;
    N = X + N;
    M = X;
    post (M=N+L and N=2*N+L)
}
```
Abstraction

- The **Precondition** must express the constraints on inputs to the function (\(L, \ N_0\) and global variables) only in this case. Note, \(N_0\) represents the value for \(N\) when the procedure begins (which may be distinct from the final value).

- **Post** must express the properties of the outputs (\(M\) and \(N\)) in terms of the inputs and outputs (and global or instance variables) only.

- Note: \|-- pre \{ method statements \} post\]

- The rule for methods is based on the substitutions that must occur for parameters. The main point of the rule is:

  Pre (with actual parameters substituted for formal parameters) and Post (with actual parameters substituted for formal parameters)

  \(\Rightarrow\) the result of executing a method call.
Application of Abstraction to Software Engineering

- Proving a methods correct using abstraction approach has 3 steps:
  1. Detail the method’s signature (name, modes (in, out, inout) and types of parameters) as well as pre and post conditions and global/instance variables.
  2. Verify $\vdash$ pre $\{S\}$ post, where S is the body of the method.
  3. Verify the call of the method using the arguments, and predicate describing the intended result of calling the method.

- Each of these steps has a corresponding step in “good practice” software engineering:
  1. Step 1 is equivalent to what must be done in software design - describe the syntax and functionality of each component.
  2. Determining that the designed functionality and implementation match is the purpose of unit testing.
  3. Examining the method call is done during unit, and integration test.
Uses of Hoare Logic in Software Engineering

- **Functional Testing** - pre and post can be used to identify boundary, nominal and exceptional test cases,
  - For example, the precondition $J>0$ may suggest the following inputs for test cases:
    - Boundary: $J=0$, $J=1$, $J=-1$
    - Nominal: $J = 25$
    - Exceptional: $J = -15$

- **Structural Testing** - predicate transformers can be used to generate path conditions that describe the conditions for executing a specific path through code. Any test case with input that satisfies the condition will cause the path to be executed,

- **Functional Specification** - pre and post are used to describe the functionality of modules