Formal Specification and Verification

Specifications

• Imprecise specifications can cause serious problems downstream
• Lots of interpretations even with technical-oriented natural language
  – “The value returned is the top of the stack”
    • Address on the top or its element?
  – “The grace period date for payment to be printed is one month after the due date.”
    • What if the date is January 31?
• To avoid these problems, formal specification methods are more precise and less amenable to ambiguity
Formal Specs

- Why Formalize?
  - Removes ambiguity and improves precision
  - Can verify that requirements have been met
  - Can reason about requirements and designs
    - Properties can be checked automatically
    - Test for consistency, explore consequences
  - Help visualize specifications
  - Have to become formal anyway to implement

- Why people don’t formalize
  - Lower level than other techniques; too much detail that is not known yet
  - Concentrates on consistent and correct models
    - Many real models are inconsistent, incorrect, incomplete
  - Some confusion over appropriate tools
    - Specification vs. modeling
    - Advocates get attached to one tool
  - Formal methods require lots of effort

Informal Specification

- Can partially circumvent natural language problems using pseudocode, flowcharts, UML diagrams, etc.
- Better than NLP, but still relies on natural language for labels, names
- Can take lots of time to draw and there is a tendency not to update them as software evolves
Types of Formal Specs

- **Model-Oriented**
  - Describe system’s behavior in terms of mathematical structures
    - Map system behavior to sets, sequences, tuples, maps
    - Use discrete mathematics to specify desired behavior

- **Property-Oriented**
  - Indirectly specify the system’s behavior by stating the properties or constraints the system must satisfy
    - **Algebraic**
      - Data type constitutes an algebra, axioms state properties of operations
    - **Axiomatic**
      - Uses predicate logic for pre/post conditions

---

Model-Oriented Specification of a Stack

- Map stack operation onto a sequence, <…x_i…>
- s’ is the stack value prior to invoking the function
- ~ is concatenation

Let stack = <…x_i…> where x_i is an int
- Invariant 0 ≤ length(stack)
- Initially stack = null_sequence

Function
- Push(s:stack, x:int)
  - Pre 0 ≤ length(s)
  - Post s = s’ ~ x
- Pop(s:stack)
  - Pre 0 < length(s)
  - Post s = leader(s’)
- Top(s: stack) returns x:int
  - Pre 0 < length(s)
  - Post x = last(s’)
Property-Oriented Specification of a Stack

- Algebraic specification
- Type IntStack
- Functions
  - Create: \( \rightarrow \) IntStack
  - Push: IntStack \( \times \) Int \( \rightarrow \) IntStack
  - Pop: IntStack \( \rightarrow \) IntStack
  - Top: \( \rightarrow \) Int
- Axioms
  - Isempty(Create) = true
  - Isempty(Push(s,i)) = false
  - Pop(Create) = Create
  - Pop(Push(s,i)) = s
  - Top(Create) = 0
  - Top(Push(s,i)) = i

Algebraic Specification of a Set

- Type: Set
- Functions
  - Create \( \rightarrow \) Set
  - Insert: Set \( \times \) Int \( \rightarrow \) Set
  - Delete: Set \( \times \) Int \( \rightarrow \) Set
  - Member: Set \( \times \) Int \( \rightarrow \) Boolean
- Axioms
  - Isempty(Create) = true
  - Isempty(Insert(s,i)) = false
  - Member(Create,i) = false
  - Member(Insert(s,i),j) = if \( (i = j) \) then true else member(s,j)
  - Delete(Create,j) = Create
  - Delete(Insert(s,j),j) = if \( (j = k) \) then delete(s,j) else
    Insert(Delete(s,k),j)
Some Formal Specs

• VDM
  – Vienna Development Method
  – Was used to formally specify the syntax and semantics of programming languages

• Z
  – Based on Zermelo-Fraenkel set theory and first order predicate logic

• See book for some details about VDM

Program Verification

• With algebraic and axiomatic specifications we may be able to formally prove that our programs are correct
  – Start with assertions that hold before our program, precondition
  – Execute some statement
  – Results in a postcondition
  – Notation: \( \{P\} S \{Q\} \)
    • \( \{P\} \) = Set of preconditions
    • \( S \) = Statement(s) executed
    • \( \{Q\} \) = Set of post conditions
Motivation

• Here is a specification:
  – void merge(int[] ArrA, int[] ArrB, int[] ArrC)

  – Requires ArrA and ArrB to be sorted arrays of the same length. C is an array that is at least as long as the length of ArrA + length of ArrB. C is a sorted array containing all elements of ArrA and ArrB.

Motivation

• Here is an implementation

```java
int i = 0, j = 0, k = 0;
while (k < ArrA.length() + ArrB.length()) {
  if (ArrA[i] < ArrB[j]) {
    ArrC[k] = ArrA[i];
    i++;
  } else {
    ArrC[k] = ArrB[j];
    j++;
  }
  k++;
}
```

Does this program meet its specifications?
Use Predicate Logic for Pre/Post Conditions

• Expressions can be true or false
• Example:

\((x>y \land y>z) \rightarrow x > z\)

\(x = y \iff y = x\)

\(\forall x, y, z \ ((x>y) \land (y>z)) \rightarrow x > z\)

\(\forall x \ (\exists y \ (y = x + z))\) ; z is unbound, x/y bound

If all variables are bound, the formula is closed

Proof Rules

• We generally work our way backward from the desired post-condition to find the weakest pre-condition

• Strength of Preconditions
  – A Weak precondition is general; it has few constraints and is the least restrictive precondition that guarantees the post-condition
    • True is the weakest
  – A Strong precondition is specific; it has more constraints to guarantee the post-condition
    • False is the strongest

• Example: Which is weaker?

\{ b>0\} \quad \{b > 10\}  
\quad a=b+1 \quad a=b+1  
\quad \{a>1\} \quad \{a>1\}
Program Correctness

• If we write formal specs we can prove that a program meets its specifications
• Program correctness only makes sense in relation to a specification
• To prove a program is correct:
  – Prove the post-condition is true after executing the program assuming the pre-condition is true
  – Apply rules working backward line by line

Proof Rules

• Proof rules help us find the weakest preconditions for each programming construct
• Proof Rule for Assignment
  – \{P\} x=e; \{Q\}
  – To find \{P\} from \{Q\} the weakest precondition is \{Q\} with all free occurrences of x replaced by e
• Proof Rule for Sequence
  – \{P\} S1; S2; \{Q\}
  – To find \{P\} from \{Q\} first find \{R\}, the weakest precondition for S2. The weakest precondition for the sequence is then found recursively \{P\} S1 \{R\}
Hoare Notation

- Can express proof rules using Hoare notation

\[ \frac{claim1,claim2...}{conclusion} \]

- This means “if claim1 and claim2 are both proven true, then conclusion must be true”

- For sequence:

\[ \frac{\{Pre\}S1\{Q\}, \{Q\}S2\{Post\}}{\{Pre\}S1; S2\{Post\}} \]

- For if-statement:

\[ \frac{\{Pre ^ c\}S1\{Post\}, \{Pre ^ Not(c)\}S2\{Post\}}{\{Pre\}if \ (c) \ then \ S1 \ else \ S2 \{Post\}} \]

Show Precondition \( \Rightarrow \) Weakest Precondition:

\( \{Pre \wedge c \rightarrow \text{Pre-for-S1}\} \) and \( \{Pre \wedge \text{not}(c) \rightarrow \text{Pre-for-S2}\} \)

---

Proving an If Statement

\[ \{ \text{true} \} \]

If \((x > y)\) then

- \(max = x;\)
- \(max = y;\)

\(\{ (max = x \lor max = y) \land (max \geq x \land max \geq y) \} \)

The then branch:

- (?)
- \(max = x;\)
- \(max = x;\)
- \( (max = x \lor max = y) \land (max \geq x \land max \geq y) \)
- Substitute \(x\) for \(max\) backwards:
- \( (x = x \lor x = y) \land (x \geq x \land x \geq y) \)
- \( (true) \lor x = y \land (true \land x \geq y) \)
- \( (x \geq y) \)
- Which is Okay since \( (Pre \wedge c) \rightarrow (x \geq y) \)

The else branch:

- (?)
- \(max = y;\)
- \(max = y;\)
- \( (max = x \lor max = y) \land (max \geq x \land max \geq y) \)
- Substitute \(y\) for \(max\) backwards:
- \( (y = x \lor y = y) \land (y \geq x \land y \geq y) \)
- \( (y = x \land \text{true}) \land (y \geq x \land \text{true}) \)
- \( (y \geq x) \)
- Which is Okay since \( (Pre \wedge \text{not}(c)) \rightarrow (y \geq x) \)

\( (true \land \text{not}(x > y)) \rightarrow (x \geq y) \)
Loops

• The Hoare rule for loops:

\[ \{c \land P\} \text{body} \{P\} \]

\[ \{P\} \text{while} (c) \text{ body} \{ \neg c \land P \} \]

P is a loop invariant; an assertion that is true throughout the loop construct.

There is no known algorithm to find loop invariants, one must be “clever”

Loop Example

• Given the short program to sum n numbers:

Original Code:

```plaintext
sum = 0;
i = 0;
while (i <= n)
{
    sum = sum + a[i];
i++;
}
```

Insert post-conditions, loop invariant:

```plaintext
{n > 0}
sum = 0;
i = 1;
{sum = 0 \land i = 1 \land n > 0}
{1 \leq i \land i \leq (n+1) \land sum = \sum(j=1,i-1)(a[j])} \text{while} (i <= n)
{
    sum = sum + a[i];
i++;
}
{sum = \sum(j=1,n)(a[j])}
```
Loop Example

• Can we show:

{\text{sum = 0} \land i = 1 \land n > 0} \rightarrow \{1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1,i-1}(a[j])\}

Substitute in 0 for sum, 1 for i:
1 \leq 1 \text{ true}
1 \leq (n+1) \text{ true since } n > 0
0 = \sum_{j=1,0}(a[j]) \text{ is vacuously true}

So we can focus on the following:

{1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1,i}(a[j])}
while \,(i \leq n)\
\{
\begin{align*}
\text{sum} &= \text{sum} + a[i]; \\
i++;
\end{align*}
\}
\{\text{sum} = \sum_{j=1,n}(a[j])\}

\text{while (i <= n)}
\
\{\text{sum} = \sum_{j=1,n}(a[j])\}

Loop Example

• The loop rule gives us: \quad \{c \land P\}body\{P\} \quad \{P\}\text{while (c)}body\{\neg c \land P\}

This means at the end of the loop we should have:
\neg c \land P
Which is:
\{i>n \land 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1,i-1}(a[j])\}

{1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1,i-1}(a[j])}
while \,(i \leq n)\
\{
\begin{align*}
\text{sum} &= \text{sum} + a[i]; \\
i++;
\end{align*}
\}
\{i>n \land 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1,i-1}(a[j])\}
\{\text{sum} = \sum_{j=1,n}(a[j])\}
Loop Example

• Show end of loop:

\[
\begin{align*}
\{ & i \geq n \land 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1}^{i-1}(a[j]) \} \Rightarrow \\
\{ & \text{sum} = \sum_{j=1}^{n}(a[j]) \}
\end{align*}
\]

Since \( i > n \) and \( i \leq n+1 \), then \( i = n+1 \)
Sum = \( \sum_{j=1}^{n+1-1}(a[j]) \)
\( \Rightarrow \) Sum = \( \sum_{j=1}^{n}(a[j]) \)

This is assuming the loop rule condition holds, which we haven’t shown yet

Loop Example

• The loop rule body:

\[
\begin{align*}
\{ & c \land P \}\text{body}\{P\} \\
\{ & P\}\text{while } (c) \text{ body}\{\neg c \land P\}
\end{align*}
\]

\[
\begin{align*}
\{ & i \leq n \land 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1}^{i-1}(a[j]) \\
& \text{sum} = \text{sum} + a[i]; \\
& i++; \\
& \{ & 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1}^{i-1}(a[j]) \}
\end{align*}
\]

Substitute backwards:

\[
\begin{align*}
\{ & i \leq n \land 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1}^{i-1}(a[j]) \\
& \{ & 1 \leq i+1 \land i+1 \leq (n+1) \land \text{sum+}a[i] = \sum_{j=1}^{i}(a[j]) \\
& \text{sum} = \text{sum} + a[i]; \\
& \{ & 1 \leq i+1 \land i+1 \leq (n+1) \land \text{sum+}a[i] = \sum_{j=1}^{i}(a[j]) \\
& i++; \\
& \{ & 1 \leq i \land i \leq (n+1) \land \text{sum} = \sum_{j=1}^{i-1}(a[j]) \}
\end{align*}
\]
Loop Example

• Show entrance of loop body:

\[
\begin{align*}
\{ i \leq n \land 1 \leq i \land i \leq (n+1) \land \sum = \sum_{j=1,i-1} a[j] \} \\
\Rightarrow \\
\{ 1 \leq i+1 \land i+1 \leq (n+1) \land \sum + a[i] = \sum_{j=1,i} a[j] \}
\end{align*}
\]

1 \leq i+1 is true since 1 \leq i
(i+1) \leq (n+1) is true since we have i \leq n
Sum + a[i] = \sum_{j=1,i} a[j] can become \sum = \sum_{j=1,i} a[j] - a[i]

This follows from sum = \sum_{j=1,i-1} a[j]]

We have now proven all of the pieces of the code!
We can continue in confidence it actually computes
the sum (we should also prove the invariant)

Practicalities

• Program proofs are currently not widely used
  – Tedious to construct
  – Can be longer than the programs they refer to
  – Can contain mistakes too
  – Requires math
  – Does not ensure against hardware errors, compiler errors, etc.
  – Only prove functional correctness, not termination, efficiency,
    etc.

• Practical formal methods:
  – Use for small parts of the program, e.g. safety-critical
  – Use to reason about changes to a program
  – Use with proof checking tools and theorem provers to automate
  – Use to test properties of the specs
Other Approaches

• Model-checking
  – A model checker takes a state-machine model and a temporal logic property and tells you whether the property holds in the model
  – temporal logic adds modal operators to propositional logic:
    – e.g. □ x  x is true now and always (in the future)
    – e.g. ◊ x  x is true eventually (in the future)
• The model may be:
  – of the program itself (each statement is a ‘state’)
  – an abstraction of the program
  – a model of the specification
  – a model of the domain
• Model checking works by searching all the paths through the state space
  – with AI techniques for reducing the size of the search
• Model checking does not guarantee correctness
  – it only tells you about the properties you ask about
  – it may not be able to search the entire state space (too big!)
  – but is (generally) more practical than proofs of correctness.