Formal Verification

Floyd- and Hoare-style reasoning

Algebraic specification and verification

Two-tier specifications

CIS 610, W98 / M Young 1/26/98 1

Outline:

* Floyd- and Hoare-style reasoning

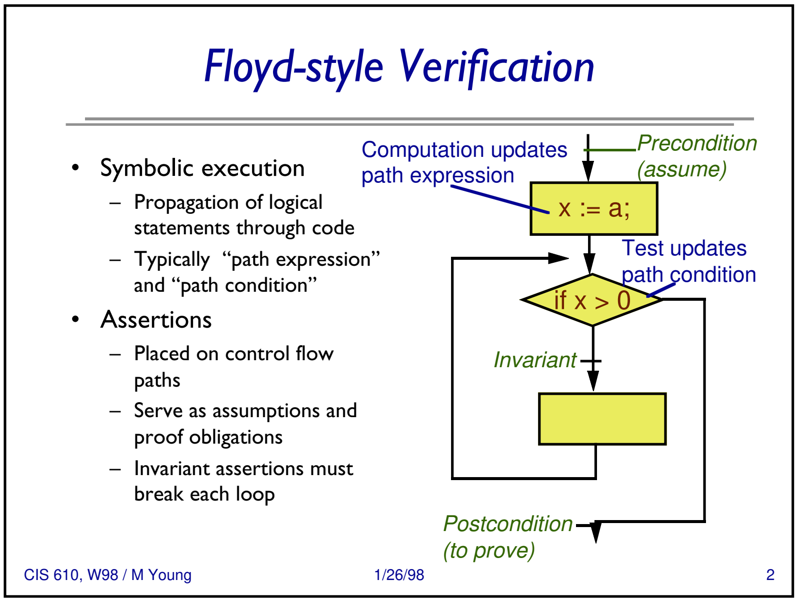
A basic technique underlying all program verification, and much program analysis, is symbolic execution. The basic technique is explained in the Hantler and King paper (which you've read by now, right?). Although program verification systems of the kind envisioned by Hantler and King have not been very successful, it is still important to understand the fundamental techniques.

* Algebraic specification

Formal verification is possible only if we have formal specs. Pre- and postcondition assertions, which form the basis for Floyd- or Hoare-style reasoning, arenot very natural for object-oriented or ADT-based components. Algebraic specifications were designed for that purpose.

* Two-tier specifications

To write clean, clear specifications, we need a notation with simple semantics. To write specifications that adequately capture the intended behavior of real program components, we need complex semantics. Two-tier specifications are just what they sound like: specifications written in two different notations, one designed for clarity and the other designed to adequately describe concrete module interfaces. They tie together algebraic specifications with Floyd- or Hoare-style reasoning about component behavior.



In the style of reasoning developed by Floyd, a program or component (e.g., procedure or method) is specified by a (precondition, postcondition) pair. Each program statement can be interpreted as a modification to a symbolic predicate. (Although this predicate is often split into a path condition and path expression, this is just a matter of convenience.) Verification consists of showing that, beginning from the precondition and assuming it to be true, every execution path to the postcondition will make the postcondition predicate true.

You can think of the path expression as a table in which symbolic expressions are associated with variables. Imperative statements (e.g., assignment statements) change the path expression, like this:

```
{ x = x0 & x0 > 3 }
y := x + 5;
{ x = x0 & y = x0 + 5 & x0 > 3 }
```

The path condition records branches:

```
{ x = x0 & y = x0 + 5 & x0 > 3 }
if (y < 2) then
    { x = x0 & y = x0 + 5 & x0 > 3 & x0+5 < 2 }
    ...
else
    { x = x0 & y = x0 + 5 & x0 > 3 & x0+5 >= 2 }
    ...
```

Since there is a potentially infinite number of control flow paths in any procedure with a loop, it is necessary to break the paths into a finite number of segments, in particular "cutting" each loop with an invariant assertion.

(Trivial) Example: Linear Search

```
public String expand(String statecode) {
        for (int i=0; i<table.length; ++i)  {
                /* assert: for all 0<=j<i, table[i].code != statecode */
                if ( table[i].code.equals(statecode) )  {
                        return table[i].name;
                }
        }
        /* assert: not exists i s.t.
         *    0<=i<table.length and table[i] = statecode
         */
        return "UNKNOWN";
}
```

First time

On loop exit

Maintaining invariant

CIS 610, W98 / M Young 1/26/98 3

Here is the simplest example I could think of to illustrate Floyd-style verification with loop-cutting assertions. I left the following header comment off the slide to save space:

```
/* expand:  from a 2-character code to a full state name,
 *   or return "UNKNOWN" if the code does not correspond to a state.
 * Precondition:  TRUE  (i.e., we don't assume anything about the input)
 * Postcondition: if exists i s.t.
 *                  0<i<=table.length and table[i].code = statecode
 *              then      expand(statecode) = table[i].name
 *              else      expand(statecode) = "UNKNOWN"
 */
public String expand(String statecode)       {   ...
```

This is slightly bogus, in that the "postcondition" is really a statement that combines pre- and postcondition, but I wanted to use natural C and Java style rather than twisting the code so that assertions could be placed in the "right" places.

There is other code defining an array of two-digit state codes and full state names, of course; I've left that off to save space.

The Floyd-style verification is performed by symbolically executing each control path from assertion to assertion (including paths from an assertion to itself). If we didn't place an "invariant" assertion in the loop, the number of paths would be infinite.

Look easy but tedious? It is, and most of the symbolic execution part can (and has) been automated. The hardest part is devising the assertions, which can be only partially automated (see the discussion of undecidability below). Simplifying expressions and proving the assertions can also be partly automated.

Exercise: Binary Search

(Do on whiteboard)


```
/* expand:  from a 2-character code to a full state name,
 *   or return "UNKNOWN" if the code does not correspond to a state.
 * Pre:  for all 0<=i<j<table.length . table[i].code <= table[j].code
 * Post: if exists i s.t. 0<i<=table.length and table[i].code = statecode
 *            then    expand(statecode) = table[i].name
 *            else    expand(statecode) = "UNKNOWN"
 */
public String expand(String statecode)      {
        int low = 0;
        int high = table.length - 1;

        while (low <= high)  {
            int i = (low + high) / 2;
            /* assert: for all 0<=j<low . table[i].code < statecode
             *      for all high<=j<table.length, table[i].code > statecode
             */
                int comparison = table[i].code.compareTo(statecode);
                if ( comparison < 0 )  {
                        low = i+1;
                } else if ( comparison > 0 ) {
                        high = i - 1;
                } else {
                        return table[i].name;
                }
        }
        /* assert: not exists i s.t.
         *    0<=i<table.length and table[i] = statecode
         */
        return "UNKNOWN";
}
```

Hoare-style Reasoning

- Block-structured rules
  - Floyd-style symbolic execution step is “add one more step”
  - Hoare-style step is “combine two blocks”

$$\frac{P \{ S \} Q,\ Q \{ T \} R}{P \{ S; T \} R} \qquad \frac{P \,\&\, Q \{ S \} Q}{P \{ \text{while (c)}\ S \} Q \,\&\, \text{not c}}$$

C.A.R. Hoare introduced a style of reasoning based on block structure, rather than following each control flow path. This style is often called “axiomatic.” The basic symbolic execution rules for individual statements are the axioms, and the rules for combining chunks into larger and larger pieces are inference rules of the form

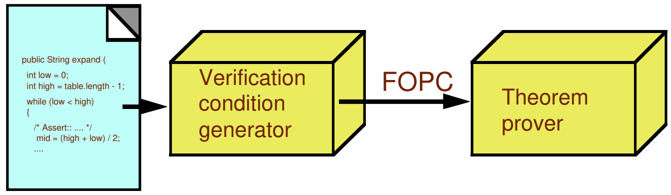
if these are true

----

then this must be true

Hoare-style reasoning is more “structured” than Floyd-style reasoning, but for our current purposes it doesn’t much matter.

Verification System Architecture



- Typical architecture (circa 1980):
  - Symbolic execution (VCG) creates logical statements to be proved (one for each assertion/assertion path)
  - General FOPC theorem prover attempts to prove each statement

CIS 610, W98 / M Young 1/26/98 6

Several automated program verification systems were built from the mid 70's onward. Typically, such a system is divided into two main parts. A verification condition generator, or vcg, performs symbolic execution to compute path expression (PE) and path conditions (PC) for each path from an assertion to the next assertion, which is the postcondition. Then it forms the "proof obligation" PC&PE => postcondition. This statement is (usually) in first-order predicate calculus (FOPC), and can be passed on to a general-purpose theorem prover.

The point of this architecture is to separate parts dependent on a particular programming language from the theorem prover. The VCG is highly dependent on a particular programming language, which determines how symbolic execution is carried forward. In principle a VCG for a particular programming language is fairly simple to build, provided one has a good programming language semantics to work from. In practice, formal semantic definitions of full programming languages are rarely available, so most symbolic executors and VCGs have processed only restricted language subsets.

Theorem provers are much more complex than symbolic executors, but over the past 20 years they have become progressively more powerful. The best theorem provers available today, such as PVS and Nqthm (formerly the Boyer-Moore prover) have produced fully checked proofs of subtle algorithms. Several published proofs of algorithms have been re-verified mechanically, typically uncovering errors in the original published proofs.

Despite the remarkable progress in theorem provers, you won't see many program verification systems built this way today. In fact, you won't see many program verification systems today at all. You are much more likely today to see the theorem prover applied for other purposes, like checking a specification for consistency, rather than verifying a program.

Undecidability Bites

```
int fermats_revenge (int a, int b, int c, int k) {
   if (  a^k + b^k == c^k  ) {
        /*  assert k < 3 */
   } ...
```

- Theory says: (PE & PC) => Post can be undecidable
  - Even reachability of a statement is equivalent to the halting problem
- In practice: Theorem provers require expert human assistance

CIS 610, W98 / M Young 1/26/98 7

Almost every program property of interest is, in theory at least, undecidable. Consider the simple question: "Can execution reach this point in the code?" That question is equivalent to the halting problem, even if we drastically restrict the programming language. So, in general, there will be propositions that a VCG can generate, but which a theorem prover cannot prove.

The more typical case is propositions that may not be formally undecidable, but which are certainly outside the capabilities of automatic theorem proving. Consider the verification condition that will be produced at the assertion in the snippet of code on this slide. Do you recognize it? It's called "Fermat's last theorem," and while it apparently has recently been proved, it took many mathematicians a long time to do it. The example is contrived, but it illustrates that fairly innocent looking assertions can be very hard to prove.

While today's theorem provers are remarkably good and still improving, there isn't much prospect that they will reach a point where they can routinely prove all the verification conditions arising from program verification without human assistance. A verification system today is rather like a chisel: In Michelangelo's hands, it produces awesome result, but it requires human expertise and a great deal of effort to produce more than a pile of gravel.

Prove what?

- A complete program proof requires a complete, formal specification
  - Preferably shorter and simpler than the program itself
- Seldom available in most domains
  - Cost effective for some critical systems, or critical parts, or critical properties
  - Formalizing (incomplete) specs may be cost effective even *without* program proof

Proofs of correctness, like any verification, are checks of correctnes between two descriptions, a specification and an implementation. To construct the kind of detailed program proof envisioned by researchers in the 70s and early 80s, we would need very detailed, formal specifications. These are seldom available, and for many kinds of systems a complete specification would be at least as lengthy (and no more understandable) than the source code. A word processor is an example the kind of system for which a complete formal specification is almost certainly impractical today.

On the other hand, formal specifications do make sense for some things. When I am a passenger on a Boeing 777 or Airbus A320 ("fly-by-wire" aircraft with no mechanical controls), I would like to think that the avionics control software has been very specified very carefully and thoroughly. In many cases, formal specifications are worthwhile for some parts of a system but not for everything. It might cost about the same to specify control of the video entertainment systems in the 777 as to specify control of the flight surfaces, but the consequences of failure are quite different.

Research in program verification largely died out in the 1980's, and was supplanted by research in the more general field of "formal methods." While many in the formal methods community still believe that routine program verification will someday be practical, they also argue that formal specifications can be valuable even without program proofs. Formalizing a specification (or perhaps just critical parts of it) shakes out ambiguity and contradictions, especially when the specification can be processed by tools designed for that purpose. Moreover, it is often practical to prove properties about *something*; it is just that the *something* is more often an algorithm or protocol or design than a program per se.

ADT Module Interface Specifications

The algebraic approach

Two-level specifications

The "precondition/postcondition" style of specification and verification is, by itself, not a very good fit with modern modular program design, including object-oriented programs. What is the post-condition for a method that inserts an object in a queue? The insert method doesn't return a meaningful value, and if we give a postcondition in terms of the encapsulated data structure then we are "breaking the abstraction."

Styles of specification have been developed specifically for describing abstract data types. They are a fairly good match for object-oriented design and programming, although not everything that can be encapsulated in a class or object is an ADT module. We will look primarily at the algebraic method of specifying ADTs, and more specifically at the two-tier approach pioneered by the Larch project.

Some of this material on ADT specs will surely be already familiar to most of you, but I want to go over it thoroughly enough to be clear about the two-tier approach, because I think it is important for analysis and verification.

Larch separates a formal, "clean" level of specifications that is good for constructing proofs, from a concrete interface level that can handle all the gory details of real interfaces. It is very hard to do both in one level of specification. I believe this is a general principal, not limited to algebraic ADT specifications --- for example, one wants the same kind of separation between a clean, formally manipulable represention and a description of concrete interfaces when dealing with concurrency (multi-threading) issues, although for concurrency different kinds of formalisms are appropriate.

ADT modules

- Abstract data types encapsulate data structures
- ADT modules are supported in
  - *object-based* languages like Ada 83, Modula 2, and CLU, which provide "packages," "modules," or "clusters"
  - *object-oriented* languages like C++, Ada 95, and Java, which add inheritance and polymorphism to "class" constructs

You already knew this, right?

Actually, object-oriented languages tend to use the “class” construct for all kinds of modules, whether or not they are abstract data types. For example, a class that provides a sorting service is probably better specified in the older procedural style than as an abstract data type. A class that manages socket communication is something else again.

Flavors of ADT specifications

- Abstract models
  - ex., stack as sequence, dictionary as set
- Algebraic specification
  - construction of special-purpose models
- Trace specifications
  - (no longer in common use; of historical interest only)

There are roughly three major approaches to specifying abstract data types.

In the abstract model approach, we use some familiar kind of structure to describe intended behavior. For example, we might say that a symbol table represents a set of items, and we might describe each operation on the symbol table in terms of set operations.

The algebraic approach is like the abstract model approach, except that instead of using "well-known" abstract models, we create the models.

There is a third kind of ADT specifications called trace specifications. Today, almost no one uses trace specifications, and we will ignore them.

The algebraic approach

Separates specification into

Syntax — also called the "signature"
  a set of functions

Semantics — axioms relating terms

Essentially like abstract model approach, but we start by defining the models

CIS 610, W98 / M Young 1/26/98 12

This is covered in the paper on Larch, which is the most well-known algebraic specification language for abstract data types.

An algebraic ADT spec contains

- One or more sorts (≈ types), one of which is usually the "type of interest"
- Constructor operations (also called generators)
- Mutator operations (also called extenders)
- Inspection operations (also called observers)
  - including an iterator for a collection type
- Axioms

Again, this is all covered in the Larch paper.

What does an algebraic spec mean?

- The meaning of a specification is a theory
- A theory is a set of terms. It includes
  - A set of built-in logical axioms
  - Axioms from the specification
  - all the formulas derivable from those axioms by the rules of logic
  - and nothing else
- The theory of queues includes Front(Insert(Insert(Insert(Create,i),j),k) =i

... and again

Absent from Queue Specification

- Meaning of Front(Delete(Q,i))
  - because Create and Insert are enough to describe all legal queues; we say the sort is generated by these two operations. All other operations, like Delete, can be removed from legal Queue terms by applying axioms as rewrite rules.
- Meaning of Front(Create())
  - we could introduce an error term, but our specifications would be more complex

This slide really has two very different points.

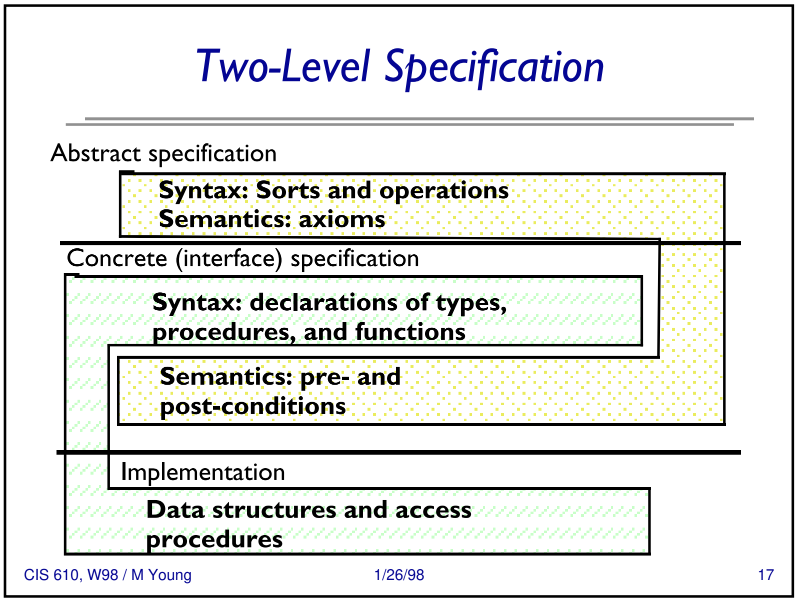
The reason Front(Delete(Q,i)) is not given a meaning is that "Delete(Q,i)" can be re-written to a simpler expression; there is a canonical form in which the only operations that need to be written are Create and Insert.

The second point is more important for our purposes --- dealing with errors (undefined terms) in an algebraic specification is possible, but nasty. Two-tiered specifications (next slide) allow us to leave this undefined in the abstract model, and handle it instead at the concrete specification level.

Two-Level Specifications: Sorts and Types

- Sort is an abstract class of objects and operations with properties. Corresponds to ...
  - Type in a programming language
    - type in Pascal or C; private type in Ada
    - class in C++, Modula3, Java;
- Representation is a concrete data structure, distinguished both from type and from sort.
- The abstract specification describes sorts, the interface specification describes coresponding types

... more from the Larch paper

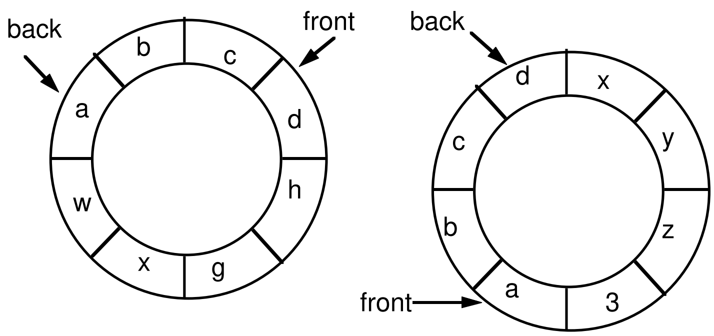


This says essentially the same thing as the diagram on page 25 of the Larch paper. In terms of analysis & testing, the essential point is that it tells us where to get pre- and post-conditions for operations (methods). A precondition associates concrete variables with abstract objects. A postcondition associates the output results with abstract objects again, and describes the relation between the "before" objects and the "after" objects.

Relation of sorts to representations

- Abstraction function:
  - abs: concrete —> abstract
  - Must be a function, but need not have an inverse
  - Example: Ring buffer (data structure) to queue (ADT)
- Structural invariant: Properties of data structure that are preserved by all operations

Representation function: Ring Buffer



abs(Q1) = abs(Q2) =
Insert(Insert(Insert(Insert(Create( ),d),c),b),a)

Pre- and Post-conditions

- Pre- and post-conditions specify interface behavior by reference to operations in algebraic specification.
- Abstraction function is the glue between algebraic specification and interface.

```
procedure Remove(R: in out RingBuffer;
                 e: out      elem_type);
    --* Requires: Abs(R) /= Create( )
    --* Ensures:  Abs(R) = Delete(Abs(R'))
    --*                 and e = Front(Abs(R'))
```

The Two-Level Approach

- Top level is (relatively) easy to reason about
  - e.g., the Larch prover can verify many properties about an LSL trait
- Interface specs are concise
  - pre- and postconditions are simplified because they refer to abstract spec

Is there a general principle here?