REQUIREMENTS ANALYSIS

Typical life stages of development:
- Requirements
- Specifications
- Top level design (often called architecture)
- Detailed design
- Code and unit test
- Integration testing

Goal now is first two of these.

What versus how

Each stage defines a transformation:
- **Specification → Implementation**
- **What to build → How to build it**
- **Requirements → Specification**
- **Specification → Design**
- **Design → Code**

Software development is the process of converting "whats" to "hows".

Process is a series of transformations as you need to show the equivalence of the "how" to its defining "what."

But: Requirements are a "how" with no "what."

Requirements Process

– Customer and designer must develop a shared understanding of what to build
– Team members not meeting customer must develop own understanding of problem
– All team members need to develop common understanding
– Organization (beyond developers) must also understand problem

Requirements process - II

Since we don't have a clear notation (a "what"), the process is often cyclical iterating to get desired solution:

– Requirements documentation – stakeholders (e.g., customers) write down proposed requirements. This can be a detailed document or one or two pages of "goals."

Techniques:

– Review existing requirements document of new system or previous systems that were similar
– Scenario analysis – a proposed use of the system – use cases (used with OO design), scripts, action tables

1. Requirements discussion

Stakeholders challenge proposed requirements:

– Rapid prototyping a method for evaluating scenarios.
– Questions asked:
  - "What is (meant by …)"
  - "How to (do …)"
  - "Who (is the agent to …)"
  - "What kinds of (…) will do "
  - "When (will …)"
  - "What relationship (between req 1 and 2)"
  - "What if (something went wrong …)"

2. Requirements evolution

Stakeholders change requirements based upon discussion:

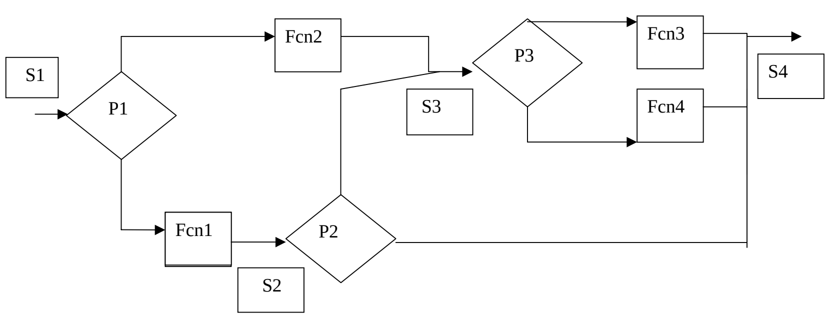
– Challenges assumptions made by group.
– Trace changes to discussion items. Traceability a major issue in system design.
– Changes:
  - Mutation – a change or addition to the requirements
  - Restriction – a change that restricts possible actions, removes ambiguity
  - Editorial – a rewrite or rewording of an existing requirement. Grammatical changes

Requirements Issues

- No good precise notation. English often used. English often ambiguous.
- Concept often not clear with customer
- After a discussion of requirements issues:
  - will look at more formal specification models
  - will look at slightly less formal specification notations
  - will look at cleanroom development

Formal Specifications

Every program can be represented by a flowchart:



Basic mathematical model

Signatures: function: domain → range

- fun1: S1 and P1 → S3
- fun2: S1 and not(P1) → S3
- fun3: S3 and not(P3) → S4
- fun4: S3 and P3 → S4
- S2 and P2 == S4
- S2 and not(P2) == S3

A program is then some complex function C:

C(fun1, fun2, fun3, fun4, p1, p2, p3):S1→S4

If we could derive these relationships formally, then we have a mathematical proof that given S1 we would HAVE to end with S4.

Problem: Such proofs are hard.

Proof Models

Axiomatic verification (closest to previous figure) – Tony Hoare – 1969. Each program statement obeys a formal axiomatic definition. All models following this approach, more or less. This is the fundamental formal method.

Weakest precondition – Edsgar Dijkstra – 1972. Similar to axiomatic verification, but allows for some non-determinism.

Algebraic data types – John Guttag – 1975. Formal specifications obey algebraic properties.

Functional correctness – Harlan Mills – 1975. Formalize symbolic execution. Execution traces. Often part of cleanroom development.

Others: Model checking, Petri nets, State machines, …

Proof Models - II

As we will see, all are hard to use in real systems. However, this formalism is a goal worth striving for in order to eliminate the ambiguity of alternative methods. Practical formal methods are derived from these models.

We will briefly look at 2 such models as examples of formal systems:

- axiomatic verification
- algebraic data types

Scalability of Formal Methods

Small grain methods (e.g., axiomatic, statement oriented) have a mathematical basis but are hard to scale up, and are hard to handle changes.

Large grain methods involve module composition allow for simpler specifications, but are less precise. VDM and Z can be considered large grain methods

Huge grain methods are typically systems. They may involve system-level protocols, such as TCP/IP, web browsers (HTML), …

Terminology

Validation – Demonstrating that a program meets its specifications by testing the program against a series of data sets

Verification – Demonstrating that a program meets its specifications by a formal argument based upon the structure of the program.

Validation assumes an execution of the program; verification generally does not.

AXIOMATIC VERIFICATION

{P1}S{P2} means that if P1 is true and S executes, then if S terminates, P2 will be true.

$\frac{A}{B}$ Means if A is true then you can state that B is true.

Axioms of Logic

Composition {P}S1{Q},{Q}S2{R}
{P}S1;S2{R}

Consequence1 {P}S{R}, R→Q
{P}S{Q}

Consequence 2 P→R, {R}S{Q}
{P}S{Q}

These embed programming constructs into mathematical logic.

Statement axioms

Conditional 1 {P^B}S{Q}, P^not(B)→Q
{P}if B then S{Q}

Conditional 2 {P^B}S1{Q},{P^not(B)S2{Q}
{P}if B then S1 else S2{Q}

While {P^B}S{P}
{P}while B do S{P ^ not(B)} where P is an invariant

Assignment {P(e)}x:=e{P(x)} where P(x) is some predicate in x.

AXIOMATIC PROGRAM PROOF

Given program: s1; s2; s3; s4; … sn and specifications P and Q:

- P is the precondition
- Q is the postcondition

Show that {P}s1;…sn{Q} by showing:

- {P}s1{p1}
- {p1}s2{p2}
- … {pn-1}sn{Q}

Repeated applications of composition:

- {P}s1; …; sn{Q}

Example: Prove the following

```
{B≥0}
X:=B
Y:=0
while X>0 do
        begin
        Y:= Y+A
        X:= X-1
end
{Y=AB}
```

Proof Outline

General method is to work backwards.

Look for loop invariant first:

- Y is part of partial product, and X is count of what remains
- So Y and XA should be the product needed, i.e., invariant
- Y+XA = AB is proposed invariant
- Try: (Y+XA=AB and X≥0)

Proof - Steps 1-4

{(Y+(X-1)A=AB and X-1≥0)} X:=X-1 {(Y+XA=AB and X≥0)} Axiom of assignment (6)

{((Y+A)+(X-1)A=AB and X-1≥0)} Y:=Y+A {(Y+(X-1)A=AB and X-1≥0)} Axiom of assignment (5)

{(Y+A)+(X-1)A=AB and X-1≥0)} Y:=Y+A; X:=X-1 {(Y+XA=AB and X≥0)} Axiom of composition (5,6)

Y+XA=AB and X>0 ➔ ((Y+A)+(X-1)A=AB and X-1≥0) Mathematical Theorem

{ Y+XA=AB and X>0} Y:=Y+A; X:=X-1 {(Y+XA=AB and X≥0)} Axiom of consequence

Proof - Steps 1-4

{(Y+(X-1)A=AB and X-1≥0)} X:=X-1 {(Y+XA=AB and X≥0)} Axiom of assignment (6)

{((Y+A)+(X-1)A=AB and X-1≥0)} Y:=Y+A {(Y+(X-1)A=AB and X-1≥0)} Axiom of assignment (5)

{(Y+A)+(X-1)A=AB and X-1≥0)} Y:=Y+A; X:=X-1 {(Y+XA=AB and X≥0)} Axiom of composition (5,6)

Y+XA=AB and X>0 ➔ ((Y+A)+(X-1)A=AB and X-1≥0) Mathematical Theorem

{ Y+XA=AB and X>0} Y:=Y+A; X:=X-1 {(Y+XA=AB and X≥0)} Axiom of consequence

Proof steps 5-8

(Y+XA=AB and X≥0) and (X>0) ➔ Y+XA=AB and X>0 Mathematical theorem

{(Y+XA=AB and X≥0)and (X>0)} Y:=Y+A; X:=X-1 {(Y+XA=AB and X≥0)}** Axiom of consequence

**

{Y+XA=AB and X≥0}while X>0 do
Y:=Y+A; X:=X-1 {(Y+XA=AB and X≥0)
and not(X>0)} While axiom

(Y+XA=AB and X≥0) and not(X>0)
➔(Y+XA=AB and X=0) ➔Y=AB Mathematical theorem

Proof - Steps 9-12

{Y+XA=AB and X≥0}while X>0 do Y:=Y+A; X:=X-1
{Y+AB} Axiom of consequence

{0+XA=AB and X≥0} Y:=0 {Y+XA=AB and X≥0}
Axiom of assignment

{0+BA=AB and B≥0} X:=B {0+XA=AB and X≥0)}
Axiom of assignment

{0+BA=AB and B≥0} X:=B; Y:=0 {Y+XA=AB and X≥0)}
Axiom of composition

Proof - Steps 13-15

(B≥0) ➔ 0+BA=AB and B≥0 Mathematical theorem

{ B≥0} X:=B; Y:=0 {Y+XA=AB and X≥0)}
Axiom of consequence

{ B≥0} program { Y=AB} Composition

Summary – Axiomatic verification

Need to develop axioms for all language features. Can do it for simple arrays, procedure calls, parameters

- Difficult, even on small programs
- Very hard to scale up to larger programs
- Does not translate well to non-mathematical problems
- Extremely expensive to implement
- Hard to automate. Manual process leads to many errors.
- A cult following – "Every program must formally verified"

BUT

- Precise – clearly delineates the "what" with the "how"
- Basis for most other verification methods, including semiformal specification notations.
- For critical applications, it may be worth the cost.

Algebraic data types

- Specification of a function, not implementation.
- Closely tied to abstract data type issues.

- Example: stacks
- Functions: newstack, push, pop, top, empty, size

Axioms

- **pop(newstack) = newstack**
- **pop(push(s,i)) = s**
- **top(newstack) = undefined**
- **top(push(s,i)) = i**
- **empty(newstack) = true**
- **empty(push(s,i)) = false**
- **size(newstack) = 0**
- **size(push(s,i)) = size(s)+1**

newstack is a constructor, push is a generator, others operators.

Heuristic – combine each constructor and generator with each operator to get axioms.

We say that any implementation of the given functions that preserves these relationships is a valid implementation of a stack.

Proof methods – Data type induction

Similar to mathematical induction

- Let f be constructor, g generator, p predicate
- Show p(f) is true (base case)
- Show p(s) ➔ p(g(s))

Then true for all s.

Theorem: size(push(s,x))>size(s)

P(s) == size(push(s,x))>size(s)

Data type induction proof

(Base case) s = newstack (Read bottom up)

Show: Size(push(newstack,i)) > size(newstack)

Size(push(newstack,i)) = size(newstack)+1 Axiom 8

Size(newstack)+1 = 0 + 1 = 1 Axiom 7

Size(push(newstack,i)) > size(newstack)

Mathematical theorem, Axiom 7

(Generating case) s=push(s', x)

Size(push(s,x))>size(s) Goal

Size(push(push(s',x),x))> size(push(s',x))

Definition of s

Size(push(s',x))+1> size(push(s',x) Axiom 8

1+Y>Y Theorem

Example 2 -- Addition

Axioms:

- Add(0,x) =x
- Add(succ(x),y) = succ(add(x,y))

0-constructor, succ-generator, add-operator

Theorem: 1+1=2 (Note: 1=succ(0), 2=succ(1))

Add(succ(0),succ(0))= succ(add(0,succ(0)))	Axiom 2
Add(succ(0),succ(0))= succ(succ(0))	Axiom 1
1+1=2	Substitution

Example 3 - Associativity

add(add(x,y),z) = add(x,add(y,z))

Proof: Base case:

Add(add(0,y),z)=add(0,add(y,z))	Goal
Add(y,z)=add(y,z)	Axiom 1

Inductive case: Add(add(x,y),z)=add(x,add(y,z))

Add(add(succ(x),y),z)= add(succ(add(x,y)),z)	Axiom 2
Add(add(succ(x),y),z)= succ(add(add(x,y),z))	Axiom 2
Add(add(succ(x),y),z)= succ(add(x,add(y,z)))	Ind. Hyp.
Add(add(succ(x),y),z)= add(succ(x),add(y,z))	Axiom 2