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.


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.

A
B  Means if A is true then you can state that B is true.
### Axioms of Logic

**Composition**

\[ \{P\}S_1\{Q\}, \{Q\}S_2\{R\} \]

\[ \{P\}S_1;S_2\{R\} \]

**Consequence 1**

\[ \{P\}S\{R\}, R \Rightarrow Q \]

\[ \{P\}S\{Q\} \]

**Consequence 2**

\[ P \Rightarrow R, \{R\}S\{Q\} \]

\[ \{P\}S\{Q\} \]

These embed programming constructs into mathematical logic.

---

### Statement Axioms

**Conditional 1**

\[ \{P \land B\}S\{Q\}, P \land \neg(B) \Rightarrow Q \]

\[ \{P\} \text{if } B \text{ then } S\{Q\} \]

**Conditional 2**

\[ \{P \land B\}S_1\{Q\}, \{P \land \neg(B)\}S_2\{Q\} \]

\[ \{P\} \text{if } B \text{ then } S_1 \text{ else } S_2\{Q\} \]

**While**

\[ \{P \land B\}S\{P\} \]

\[ \{P\} \text{while } B \text{ do } S\{P \land \neg(B)\} \text{ where } P \text{ is an invariant} \]

**Assignment**

\[ \{P(e)\}x:=e(P(x)) \text{ where } P(x) \text{ is some predicate in } x. \]
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}
1 X := B
2 Y := 0
3 while X > 0 do
4      begin
5        Y := Y + A
6        X := X - 1
7      end
8    {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: \( (Y + XA = AB \text{ and } X \geq 0) \)

Proof - Steps 1-4

\[
\{(Y + (X-1)A = AB \text{ and } X-1 \geq 0)\} \quad X := X - 1 \quad \{(Y + XA = AB \text{ and } X \geq 0)\}
\]

Axiom of assignment (6)

\[
\{(Y + A) + (X-1)A = AB \text{ and } X-1 \geq 0\} \quad Y := Y + A \quad \{(Y + (X-1)A = AB \text{ and } X-1 \geq 0)\}
\]

Axiom of assignment (5)

\[
\{(Y + A) + (X-1)A = AB \text{ and } X-1 \geq 0\} \quad Y := Y + A \quad X := X - 1
\quad \{(Y + XA = AB \text{ and } X \geq 0)\}
\]

Axiom of composition (5, 6)

\( Y + XA = AB \text{ and } X > 0 \implies ((Y + A) + (X-1)A = AB \text{ and } X-1 \geq 0) \)

Mathematical Theorem

\[
\{ Y + XA = AB \text{ and } X > 0 \} \quad Y := Y + A \quad X := X - 1 \quad \{(Y + XA = AB \text{ and } X \geq 0)\}
\]

Axiom of consequence
Proof - Steps 1-4

\[
\{(Y+(X-1)A=AB \text{ and } X-1\geq 0)\} \quad X:=X-1 \quad \{(Y+XA=AB \text{ and } X\geq 0)\}
\]

Axiom of assignment (6)

\[
\{(Y+A)+(X-1)A=AB \text{ and } X-1\geq 0)\} \quad Y:=Y+A \quad \{(Y+(X-1)A=AB \text{ and } X-1\geq 0)\}
\]

Axiom of assignment (5)

\[
\{(Y+A)+(X-1)A=AB \text{ and } X-1\geq 0)\} \quad Y:=Y+A; \quad X:=X-1
\]

\[
\{(Y+XA=AB \text{ and } X\geq 0)\}
\]

Axiom of composition (5,6)

\[
Y+XA=AB \text{ and } X>0 \Rightarrow ((Y+A)+(X-1)A=AB \text{ and } X-1\geq 0)
\]

Mathematical Theorem

\[
\{(Y+XA=AB \text{ and } X\geq 0)\}
\]

Axiom of consequence

Proof steps 5-8

\[
(Y+XA=AB \text{ and } X\geq 0) \text{ and } (X>0) \Rightarrow Y+XA=AB \text{ and } X>0
\]

Mathematical theorem

\[
\{(Y+XA=AB \text{ and } X\geq 0)\}\text{ and } (X>0)) \quad Y:=Y+A; \quad X:=X-1
\]

\[
\{(Y+XA=AB \text{ and } X\geq 0)\}
\]

Axiom of consequence

**

\[
Y+XA=AB \text{ and } X\geq 0\}
\]

while $X>0$ do

\[
Y:=Y+A; \quad X:=X-1 \quad \{(Y+XA=AB \text{ and } X\geq 0)\}
\]

and not($X>0$)

While axiom

\[
(Y+XA=AB \text{ and } X\geq 0) \text{ and not}(X>0)
\]

$\Rightarrow (Y+XA=AB \text{ and } X=0) \Rightarrow Y=AB$

Mathematical theorem
Proof - Steps 9-12

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

\{0 + XA = AB \text{ and } X \geq 0\} \quad Y := 0 \quad \{Y + XA = AB \text{ and } X \geq 0\} \quad \text{Axiom of assignment}

\{0 + BA = AB \text{ and } B \geq 0\} \quad X := B \quad \{0 + XA = AB \text{ and } X \geq 0\}\}
\text{Axiom of assignment}

\{0 + BA = AB \text{ and } B \geq 0\} \quad X := B; \quad Y := 0 \quad \{Y + XA = AB \text{ and } X \geq 0\}\}
\text{Axiom of composition}

Proof - Steps 13-15

\(B \geq 0\) \Rightarrow 0 + BA = AB \text{ and } B \geq 0 \quad \text{Mathematical theorem}

\{B \geq 0\} \quad X := B; \quad Y := 0 \quad \{Y + XA = AB \text{ and } X \geq 0\}\}
\text{Axiom of consequence}

\{B \geq 0\} \quad \text{program} \quad \{Y = AB\} \quad \text{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) \implies 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))+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))

\[
\text{Add}(\text{succ}(0),\text{succ}(0)) = \text{succ}(\text{add}(\text{succ}(0),\text{succ}(0))) \quad \text{Axiom 2} \\
\text{Add}(\text{succ}(0),\text{succ}(0)) = \text{add}(\text{add}(\text{succ}(0)),\text{succ}(0)) \quad \text{Axiom 1} \\
1+1=2 \quad \text{Substitution}
\]

Example 3 - Associativity

\[\text{add}(\text{add}(x,y),z) = \text{add}(x,\text{add}(y,z))\]

Proof: Base case:
\[
\text{Add}(\text{add}(0,y),z) = \text{add}(0,\text{add}(y,z)) \quad \text{Goal} \\
\text{Add}(y,z) = \text{add}(y,z) \quad \text{Axiom 1}
\]

Inductive case: \[\text{Add}(\text{add}(x,y),z) = \text{add}(x,\text{add}(y,z))\]
\[
\text{Add}(\text{add}(\text{succ}(x),y),z) = \text{add}(\text{add}(\text{succ}(x),y),z) \quad \text{Axiom 2} \\
\text{Add}(\text{add}(\text{succ}(x),y),z) = \text{add}(\text{add}(\text{add}(x,y),z)) \quad \text{Axiom 2} \\
\text{Add}(\text{add}(\text{succ}(x),y),z) = \text{add}(\text{add}(x,\text{add}(y,z))) \quad \text{Ind. Hyp.} \\
\text{Add}(\text{add}(\text{succ}(x),y),z) = \text{add}(\text{add}(x,\text{add}(y,z))) \quad \text{Axiom 2}
\]