Formal Specification

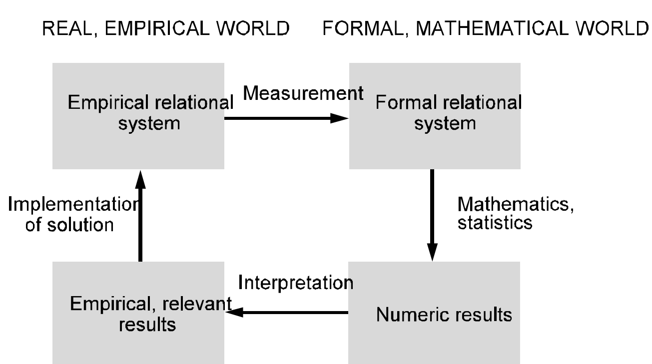
Objectives

- To explain why formal specification techniques help discover problems in system requirements
- To describe the use of algebraic techniques for interface specification
- To describe the use of model-based techniques for behavioral specification

Formal methods

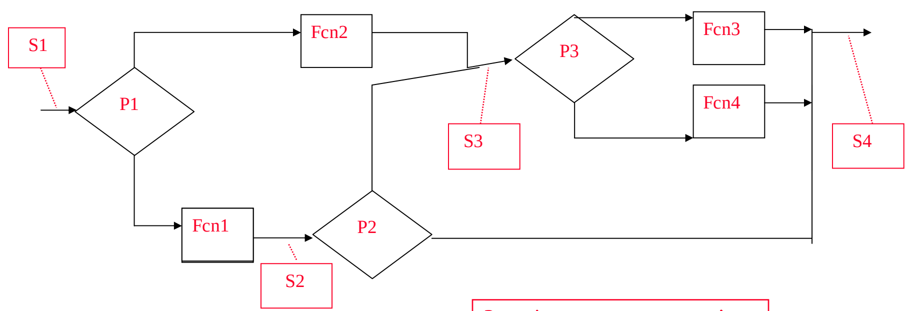
- Formal specification is part of a more general collection of techniques that are known as formal methods.
- These are all based on mathematical representation and analysis of software.
- Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.

Why formal methods?



Formal Specifications

Every program can be represented by a flowchart:



Predicate true at this point in a program:

S1 and S4 represent the specifications.

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, …

Acceptance of formal methods

- Formal methods have not become mainstream software development techniques as was once predicted
  - A. Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced;
  - B. Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market;
  - C. The scope of formal methods is limited. They are not well-suited to specifying and analyzing user interfaces and user interaction;
  - D. Formal methods are still hard to scale up to large systems.
- One of these is false, another is unfortunately true, the other two are true. Which fit into each category?

Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in critical systems engineering. There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

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.

Use of formal specification

- Formal specification involves investing more effort in the early phases of software development.
- This reduces requirements errors as it forces a detailed analysis of the requirements.
- Incompleteness and inconsistencies can be discovered and resolved.
- Hence, savings as made as the amount of rework due to requirements problems is reduced.

Cost profile

- The use of formal specification means that the cost profile of a project changes
  - There are greater up front costs as more time and effort are spent developing the specification;
  - However, implementation and validation costs should be reduced as the specification process reduces errors and ambiguities in the requirements.
  - We'll see this again with cleanroom

Example specification techniques

- Algebraic specification
  - The system is specified in terms of its operations and their relationships.
- Model-based specification
  - Specification is a mathematical proof of correctness
  - The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences. Operations are defined by modifications to the system's state.

Interface specification

- Large systems are decomposed into subsystems with well-defined interfaces between these subsystems.
- Specification of subsystem interfaces allows independent development of the different subsystems.
- Interfaces may be defined as abstract data types or object classes.
- The algebraic approach to formal specification is particularly well-suited to interface specification as it is focused on the defined operations in an object.

Specification operations

- Constructor operations. Operations which create entities of the type being specified.
- Inspection operations. Operations which evaluate entities of the type being specified.
- To specify behaviour, define the inspector operations for each constructor operation.

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

1. **pop(newstack) = newstack**
2. **pop(push(s,i)) = s**
3. **top(newstack) = undefined**
4. **top(push(s,i)) = i**
5. **empty(newstack) = true**
6. **empty(push(s,i)) = false**
7. **size(newstack) = 0**
8. **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) is defined as size(push(s,x))>size(s)

Data type induction proof

(Base case) s = newstack

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:

1. Add(0,x) =x
2. 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

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 $\dfrac{\{P\}S1\{Q\},\{Q\}S2\{R\}}{\{P\}S1;S2\{R\}}$

Consequence1 $\dfrac{\{P\}S\{R\},\ R \rightarrow Q}{\{P\}S\{Q\}}$

Consequence 2 $\dfrac{P \rightarrow R,\ \{R\}S\{Q\}}{\{P\}S\{Q\}}$

These embed programming constructs into mathematical logic.

Statement axioms

Conditional 1 — $\frac{\{P\&B\}S\{Q\},\ P\&not(B) \rightarrow Q}{\{P\}\text{if } B \text{ then } S\{Q\}}$

Conditional 2 — $\frac{\{P\&B\}S1\{Q\},\{P\&not(B)\}S2\{Q\}}{\{P\}\text{if } B \text{ then } S1 \text{ else } S2\{Q\}}$

While — $\frac{\{P\&B\}S\{P\}}{\{P\}\text{while } B \text{ do } S\{P \ \&\ not(B)\}}$
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} entire 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.

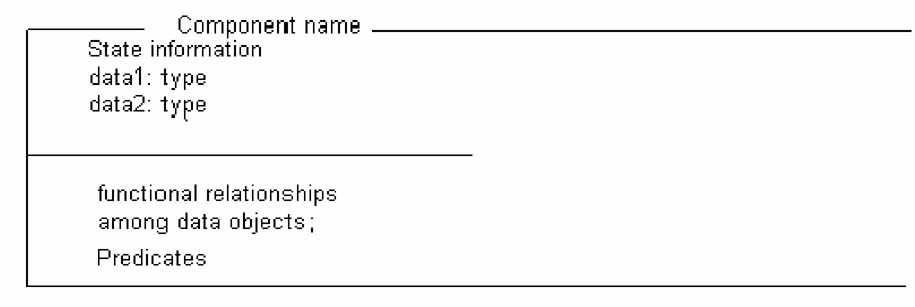
Behavioral specification

- Algebraic specification can be cumbersome when the object operations are not independent of the object state.
- Model-based specification exposes the system state and defines the operations in terms of changes to that state.
- The Z notation is a mature technique for model-based specification. It combines formal and informal description and uses graphical highlighting when presenting specifications.

Specification language Z

Defines the data, types, functions, and relationships among these in a specification

General syntax:



Some notation

- ΞX – State X doesn't change
- ΔX – State X changes
- ΘS'=ΘS – S invariant
- ℙ X – a set of X; 𝔽 X – a finite set of X
- dom – domain; ran – range
- f: X —+–> Y – a function from X to Y
- f: X >–+–> Y – a partial injection (Not every X is defined and each Y has a unique X)
- f ⊕ {X |––> Y} – f appended with f(X) = Y
- {X} ⩤ f – f with x removed from domain of f
- x' – final value of x
- x? – x is an input data item
- x! – x is an output data item

Z Example - MSWE 607 grading

ClassRegistration

ClassRoster
EnrolledInUMUC: $\mathbb{F}$ students
EnrolledInUMCP: $\mathbb{F}$ students
UMUCClasses: $\mathbb{F}$ ClassNumbers
UMCPClasses: $\mathbb{F}$ ClassNumbers
InUMUC: EnrolledInUMUC—+—>UMUCClasses
InUMCP: EnrolledInUMCP—+—>UMCPClasses
ClassList!: $\mathbb{F}$ students
UMCPClassList: $\mathbb{F}$ students

ClassList! = {x:EnrolledInUMUC | InUMUC(x) $\cap$ {607} $\neq \emptyset$}
UMCPClassList = {x:EnrolledInUMCP | InUMCP(x) $\cap$ {607} $\neq \emptyset$}
UMCPClassLIst = ClassList!

Z specification - 2

PaperGradePart

Δ ClassRoster
ClassList: $\mathbb{F}$ students
PaperGrade?: ClassList —+—>$\mathbb{N}$

PartialGrade!: ClassList —+—> {Letter |—>$\mathbb{N}$}

---

dom(PartialGrade!) = ClassList
{x: ClassList | PartialGrade!(x) = { p|—>PaperGrade?(x)}}

Z specification - 3

ExamOneGradePart

Δ ClassRoster
ClassList: $\mathbb{F}$ students
ExamOneGrade?: ClassList —+—>$\mathbb{N}$
PartialGrade!: ClassList —+—> $\mathbb{F}$ {Letter |—>$\mathbb{N}$}

{x: ClassList | PartialGrade!(x) =PartialGrade(x) $\oplus$ {e|—>ExamOneGrade?(x)}}

Z specification - 4

ExamTwoGradePart

Δ ClassRoster
ClassList: $\mathbb{F}$ students
ExamTwoGrade?: ClassList —+—>$\mathbb{N}$

{x: ClassList | PartialGrade!(x) =PartialGrade(x) $\oplus$ {t|—>ExamTwoGrade?(x)}}

Z specification - 5

ExamTwoGradePart

Δ ClassRoster
ClassList: $\mathbb{F}$ students
ExamTwoGrade?: ClassList $\rightarrow\!\!\!\!\!+\;\mathbb{N}$
PartialGrade!: ClassList $\rightarrow\!\!\!\!\!+\;\mathbb{F}$ {Letter $\mapsto \mathbb{N}$}

{x: ClassList | PartialGrade!(x) =PartialGrade(x) $\oplus$ {t $\mapsto$ ExamTwoGrade?(x)}}

Z specification - 6

FinalGradePart

ClassRoster
ClassList: $\mathbb{F}$ students
Curve: $\mathbb{R}$ $\rightarrow\!\!\!\!\!+\;$ Letter
PartialGrade: ClassList $\rightarrow\!\!\!\!\!+\;\mathbb{F}$ {Letter $\mapsto \mathbb{N}$}
FinalGrade!: ClassList $\rightarrow\!\!\!\!\!+\;$ Letter

---

dom(FinalGrade!) = ClassList
ran(FinalGrade!) = {A,B,C,F}
{x:ClassList | dom(PartialGrade(x))={p,e,t,f}
Curve(X) = $(X\geq 90 \Rightarrow A) \cup (X\geq 80 \cap X<90 \Rightarrow B) \cup (X\geq 70 \cap X<80 \Rightarrow C) \cup (X<70 \Rightarrow F)$
{x: ClassList | FinalGrade!(x) = Curve(.10*PartialGrade(x)(p) +.25*PartialGrade(x)(e)
+.25*PartialGrade(x)(t) +.40*PartialGrade(x)(f))

Z summary

- Precise description of functionality
- Have we proven the functionality correct?
  - We've replaced low level code with more abstract descriptions of the functionality
  - Need to verify these, but easier to do than the previous axiomatic example

Key points

- Formal system specification complements informal specification techniques.
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification.
- Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system.
- Formal specification techniques are most applicable in the development of critical systems and standards.

Key points

- Algebraic techniques are suited to interface specification where the interface is defined as a set of object classes.
- Model-based techniques model the system using sets and functions. This simplifies some types of behavioural specification.
- Operations are defined in a model-based spec. by defining pre and post conditions on the system state.