Practical Formal Specification Fundamentals

Formal Specification

Why even bother when building modern commercial software?

Formally describing system data structures and functions paves the way to demonstrating the correctness (or not) of systems, and allows for the possible automation of this process. “Mission-critical” systems generally contain parts that have to be formally specified to achieve or meet obligatory regulatory certification.

General Agenda

- Generic introduction to formal specification
- Formalism in Software Engineering
- Sets and specification calculi
- Algebraic specifications
- Model-based specifications

What is Formal Specification

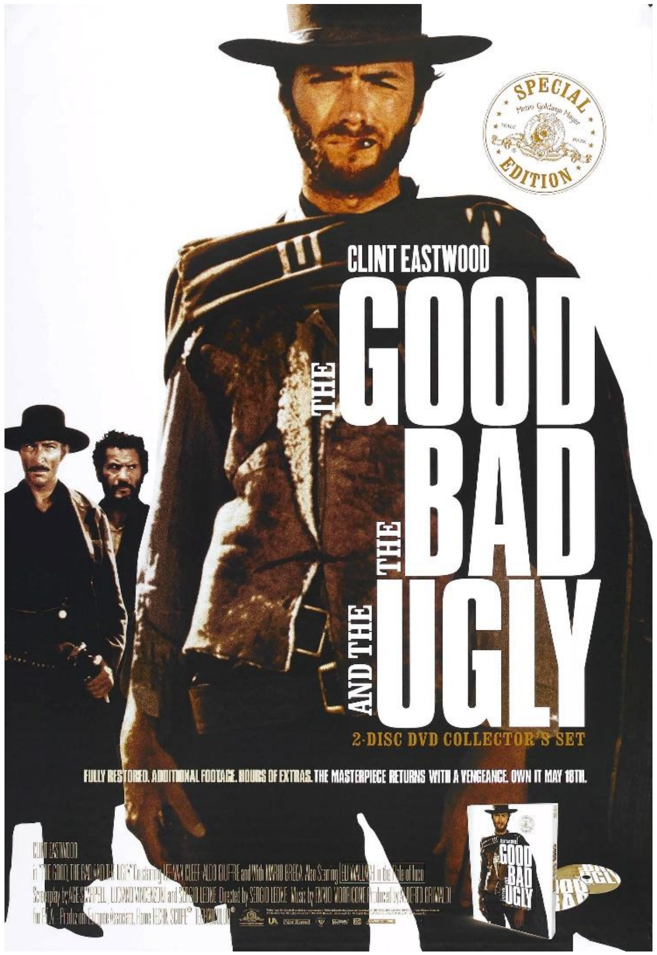
Generic introduction to formal specification

Using a description vocabulary, syntax and semantics, that have been formally defined, to specify behaviour of systems.

Therefore: The components of such a description ***must*** be based on mathematical principles!

Examples of such principles include the application of set theory, propositional and predicate calculi, algebraic specification, or their derivatives.

A well-known (cult) film


SPECIAL EDITION
CLINT EASTWOOD
THE GOOD THE BAD AND THE UGLY
2-DISC DVD COLLECTOR'S SET
FULLY RESTORED. ADDITIONAL FOOTAGE. HOURS OF EXTRAS. THE MASTERPIECE RETURNS WITH A VENGEANCE. OWN IT MAY 18TH.

The “Good”

- Using them forces one to deeply understand the behaviour of a system and its manifestation as a software solution;
- The only approach available to prove correct conformance to functional specification, and consequently correct behaviour;
- System behaviour modelled using formal specifications can be reasoned about using proven mathematical techniques and relationships;
- Can be clear indicators towards defining useful prototypes and testing scenarios;
- Can be used, or considered, as an internal prognostic (peace-of-mind) tool for developers;
- Focus-oriented – problem discovery.

The "Bad"

- Their effective use is not simply an acquired talent, but does imply an affinity to such analytical methods;
- They are labour-intensive and may increase the overall cost of the software development process;
- Many modern software development courses skirt the whole domain of formal specification – so the talent is rare;
- Formal specifications are non-compressive and directly "non-evident";
- Not applicable to every type of software solution;
- Mainstream software development deals with solutions whose benefit from formal treatment is unclear;
- Not ideal as a vehicle to convey development information to non-technical stakeholders.

The "Ugly"

- Software developers sometimes wrongly consider testing as a full or partial substitute for verification;
- Developers sometimes tend to reason about the correctness of a system through the lens of testing;
- Developers confuse validation with verification.
- It is not difficult to loosely understand formal specification and consequently loosely attribute it to a variety of specification models.
- They do not always scale up easily and evidently.

Always bear in mind…

Validation = Are we building the right solution?

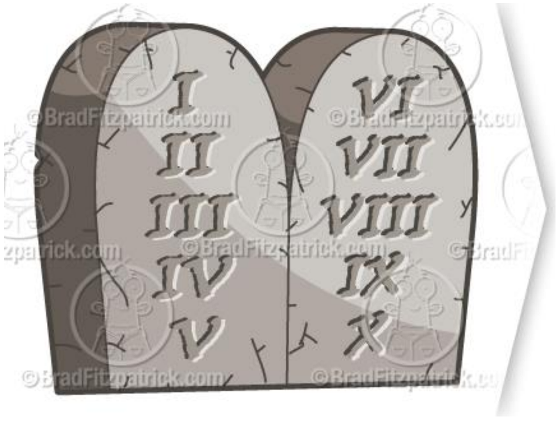
Verification = Are we building the solution right?

The 14 “myths” of formal methods in SE

1. They guarantee perfect software and eliminate the need for testing;
2. They are all about proving programs correct;
3. They are only useful in safety-critical systems;
4. Their application requires highly trained mathematicians;
5. Their applications increases development costs;
6. They are unacceptable to users;
7. They are not used on real large-scale systems;
8. They delay the development process;
9. They are not supported by tools;
10. They replace traditional engineering design methods;
11. They only apply to software;
12. They are not required [i.e. carried out as optional addition];
13. They are not supported [by the development community and its stakeholders];
14. Formal methods [specialising] people always use formal methods.

A. Hall. “Seven myths of formal methods”. In: Software, IEEE 7.5 (Sept. 1990), pp. 11–19;
J.P. Bowen and M.G. Hinchey. “Seven more myths of formal methods”. In: Software, IEEE 12.4 (1995), pp. 34–41.

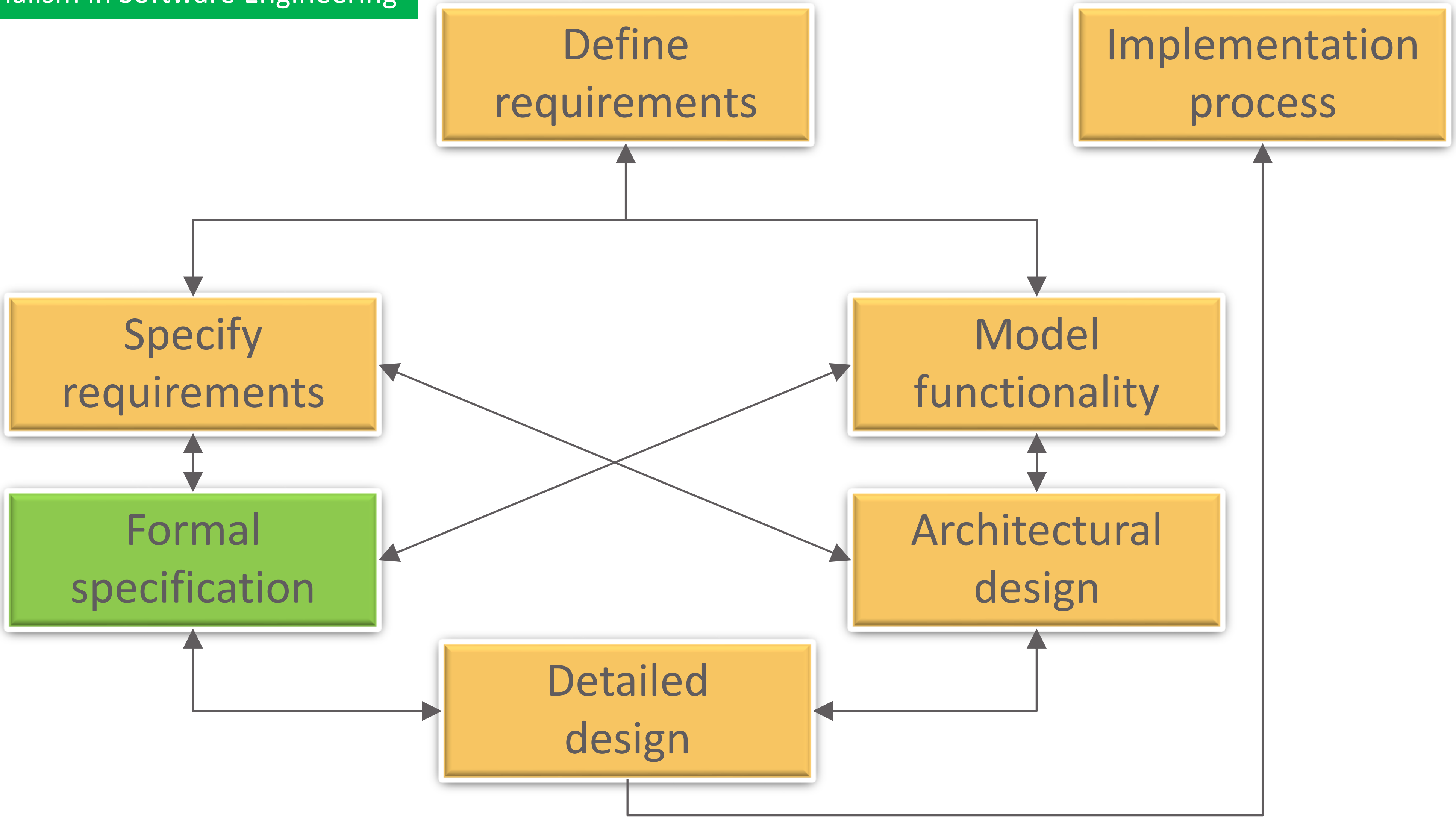
The resulting "10 Commandments"



- Thou shalt choose an appropriate notation.
- Thou shalt formalise but not over-formalise.
- Thou shalt estimate costs.
- Thou shalt have a formal methods guru on call.
- Thou shalt not abandon thy traditional development methods.
- Thou shalt document sufficiently.
- Thou shalt not compromise thy quality standards.
- Thou shalt not be dogmatic.
- Thou shalt test, test, and test again.
- Thou shalt reuse.

Jonathon P. Bowen and Michael G. Hinchey, Ten Commandments of Formal Methods, *IEEE Computer*, **28**(4):56--63, April 1995.

Position of Formal methods in SE



Categories of Formal Specification in SE

- **Algebraic**

  What can a system do; what can you expect from a system?

- **Model-based**

  How does a system move from state to state?

Example: My whiskey collection!

Looking at it Algebraically:

Which operations can I perform on my collection?

Which outcomes can I predict from such operations?

Looking at it from a Model-based perspective:

How does the state of my collection change according to operations applied to it?

How can I determine the various states of my collection?

The Basic Concepts of Formal Specification

- Set theory

  Reasoning about collections and their logical interactions

  e.g. Cars, buses, tricks or motorbikes are all forms of vehicles.

  Formally expressed as: {car,bus,truck,motorbike} $\subseteq$ *{vehicles}*

- Propositional calculus (Zeroth-order logic)

  Reasoning about statements and what can be inferred from them

  e.g. p = A lab is a room; q = AC is off in an empty room; r = AC is off.

  Formally expressed as: $p \wedge q \Rightarrow r$

- Predicate calculus (First-order logic)

  Reasoning about the properties of propositional elements as groups of elements

  e.g. "x" is a room of type "lab"; "L" is a specific empty Lab; "O" is an AC that is off; For all labs that are empty labs, their AC is off.

  Formally expressed as: $\forall x(Lx \Rightarrow Ox)$

Sets

- Are collections of elements
- Have elements that can be tangible or intangible entities
- Are represented by standard notation “{ }”
- Are manipulated by standard elementary operations
- Are entities which can interact with each other
- Can be denoted through meaningful or symbolic identifiers

Simple set examples:

Set of colours:	*{green,blue,yellow}*
Set of sports:	*{tennis,football,skating}*
Set of lecturers in this room:	*{ernest}*
Empty set:	*{ } or ∅*

Main Set Operators

If more explanation is required, operator definitions can be found in most formal specification textbooks or Internet sources.

Examples:

Valletta ∈ {Maltese towns}	*Membership*
Kiev ∉ {Maltese towns}	*Membership*
#{joe,veronica,mark} = 3	*Cardinality*
{paul,richard,claire,george} ⊆ {Group B}	*Inclusion*
P{Dynamo,CSKA} = { { },{Dynamo},{CSKA},{Dynamo,CSKA} }	*Power Set*
{ford,toyota,kia} ∪ {toyota,audi} = {ford,toyota,kia,audi}	*Union*
{ford,toyota,mercedes} ∩ {ford,audi} = {ford}	*Intersection*
{ford,toyota,subaru} \ {ford,toyota,kia} = {subaru}	*Complement*
{ford,toyota,subaru} ∆ {ford,toyota,kia} = {subaru,kia}	*Sym. Difference*
{1,2,3} **x** {2,4} = { {1,2},{2,2},{3,2},{1,4},{2,4},{3,4} }	*Cartesian prod.*

Sets using Comprehensive Specification

Comprehensive Specification of resulting sets

- Direct *(as in the previous slide)*
- Resulting from operations on other sets, as follows:

Basically, we wish to state the following to specify a set:

Create a **new set** resulting from using **the original set** whose elements **are from a specific range**, **selecting elements** according to **a specific condition** and **applying to these elements** **a specific operation**.

In constructor form

$$\{set : range \mid condition \bullet operation\}$$

In general: $\{\textbf{\textit{Signature}} \mid \textbf{\textit{Predicate}} \bullet \textbf{\textit{Term}}\}$

…or in formal notation: $\{x : X \mid P(x) \cdot E(x)\}$

Set Construction Examples

Alternate even numbers:

$\{ x : N \mid x \bmod 2 = 0 \cdot 2 * x \} = \{0,4,8,12,16,20,\ldots\}$

Tens:

$\{ x : N \mid x \cdot 10 * x \} = \{0,10,20,30,40,\ldots\}$

Squares of multiples of 4 (excluding zero):

$\{ x : Z \mid (x \bmod 4 = 0) \wedge (x > 0) \cdot x * x \} = \{16,64,144,256,\ldots\}$

Some Set-Based Specification Exercises

Write 3 elements from the sets specified by the following comprehensive specifications:

$\{n : N \mid n > 10 \wedge n < 20 \bullet n\}$	{11,12,13,…}
$\{n : N \mid n^3 > 10 \bullet n\}$	{3,4,5,…}
$\{x, y : N \mid x + y = 100 \bullet (x, y)\}$	{(0,100),(1,99),(2,98),…}
$\{x, y : N \mid x + y = 5 \bullet x^2 + y^2\}$	{25,17,13,…}

Interpret the following:

$\{m : monitors \mid MonitorState(m, on) \bullet m\}$	Monitors that are on
$\{f : SysFiles \mid f \in DelFiles \wedge f \in ArcFiles \bullet f\}$	Deleted system archive files

Write the comprehensive specification of:

$\{(10,100),(11,121),(12,144),(13,169),(14,196)\}$ $\{x{:}N \mid x \geq 10 \bullet (x,x^2)\}$

Propositional Calculus

- Deals with straightforward logical reasoning
- Basically consists of statements which can be true *or* false *(the "Excluded Middle" law)*
- Are mutually exclusive, i.e. never true or false at the same time *(the "Contradiction" law)*
- Is fundamental, i.e. forms the basis of higher-order logic
- Is axiomatic, i.e. not in itself subject to further proof
- In theory, can be used to describe anything that can be represented in the form of statements

Proposition (and non) Examples

- Some birds can fly
- The nation of Malta is in Asia
- Dogs are mammals
- All fish live in water
- All fish live in sea water
- Mary is the only lady in our group

ARE

- Sit down.
- How are you today?
- Get my tea, please.
- What is the weather like?

ARE NOT

Representing Propositions

Consider the following propositions that all make the same statement

- 10 is greater than 8
- 8 is less than 10
- 8 < 10
- 10 > 8
- There is a positive number such that if we add it to eight the result would be ten.

Propositional Calculus Notation

- Negation (not): ¬
- Conjunction (and): ∧
- Disjunction (or): ∨
- Material implication (if...then): →
- Biconditional (if and only if): ↔

Some symbolic examples:

$\neg((P \vee Q) \rightarrow Q)$ *true*

$\neg((P \wedge Q) \vee \neg R) \leftrightarrow P$ *false*

$\neg P \wedge (P \vee (Q \rightarrow P))$ *false*

$((P \rightarrow Q) \wedge (R \rightarrow S) \wedge (P \vee R)) \rightarrow (Q \vee S)$ *true*

Taking:
P as true
Q as false
R as false
S as true

Contradictions and Tautologies

- A *contradiction* is a proposition that is always false for all possible values and variables in it.
- A *tautology* is a proposition that is always true for all possible values and variables in it.

Examples:

$a \wedge \neg a$	contradiction
$a \vee \neg a$	tautology
$(a \wedge b \wedge c) \Rightarrow (c \Rightarrow a)$	tautology

De Morgan's Laws (at the heart of transformation)

$$\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$$
$$\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$$

Some other propositional calculus transformation rules regularly used:

• *Implication (IF)*	$(P \rightarrow Q) \wedge (Q \rightarrow R) \rightarrow (P \rightarrow R)$
• *Bicondition (IF AND ONLY IF)*	$(P \rightarrow Q) \wedge (Q \rightarrow P) \rightarrow (P \leftrightarrow Q)$
• *Conjunction*	$(P \wedge Q)$
• *Disjunction introduction*	$P \rightarrow (P \vee Q)$
• *Disjunctive syllogism*	$((P \vee Q) \wedge \neg P) \rightarrow Q$
• *Constructive*	$(((P \rightarrow Q) \wedge (R \rightarrow S)) \wedge (P \vee R)) \rightarrow (Q \vee S)$
• *Absorption*	$(P \rightarrow Q) \leftrightarrow (P \rightarrow (P \wedge Q))$
• *Hypothetical syllogism*	$((P \rightarrow Q) \wedge (Q \rightarrow R)) \rightarrow (P \rightarrow R)$

Using propositions to describe behaviour

Consider the following text fragment describing an aspect of the behaviour of an intruder alarm system *(adapted from Sommerville, I.)*:

“The system should be considered to be ready for intruders (alert) only when it is armed and in practice alert mode. If the system is in teaching mode and in practice alert mode, then it is considered to be alert. The system should be able to be in teaching mode and in practice alert mode while still being not alert.”

Specifying the text using Propositions

Consider the following text:

`The system should be considered to be ready for intruders (alert) only when it is armed and in practice alert mode.`

alert $\leftrightarrow$ armed $\wedge$ practice

`If the system is in teaching mode and in practice alert mode, then it is considered to be alert.`

teaching $\wedge$ practice $\rightarrow$ alert

`The system should be able to be in teaching mode and in practice alert mode while still being not alert.`

teaching $\wedge$ practice $\wedge \neg$ alert

Therefore…

Detecting Contradictions

From the previous analysed specification:

alert $\leftrightarrow$ armed $\wedge$ practice

teaching $\wedge$ practice $\rightarrow$ alert

teaching $\wedge$ practice $\wedge$ $\neg$alert

The second and third propositions yield a contradiction:

teaching $\wedge$ practice $\rightarrow$ alert

teaching $\wedge$ practice $\wedge$ $\neg$alert

alert $\wedge$ $\neg$alert *…contradiction!*

Furthermore, the first proposition yields another contradiction:

teaching $\wedge$ practice $\wedge$ $\neg$(armed $\wedge$ practice) *simplifies to…*

alert $\wedge$ $\neg$armed $\vee$ $\neg$practice *…contradiction because being alert requires being armed!*

Consider This Statement

`“Fido is a dog, dogs like bones, so Fido likes bones”.`

Propositional analysis of this sentence yields three propositions, namely:

Fido is a dog	(propos. 1) …let’s call this “P”
Dogs like bones	(propos. 2) … “Q”
Fido likes bones	(propos. 3) … “R”

Can we derive “R” from “P” and “Q” using purely propositional calculus? – Naturally, no (we can only infer it), as no properties of “P” and “Q” are known, apart from them being true or false.

Therefore…

Introducing the notion of "a predicate"

We resort to predicate calculus.

Formally, predicates can be seen as direct indicators of object properties and relationships. Denoted as $P(x)$, where P denotes the predicate on the term(s) represented by x.

- Examples of *unary* predicates:

dog(fido)	*=true*;	Df	
dog(lecturer)	*=false.*	Dl	*...(?)*

- Examples of *n-ary* predicates:

owned(Labrador,Boxer);	$O(l,b)$
father(John,Mary);	$F(j,m)$
team(Paul,Albert,Vincent).	$T(p,a,v)$

Predicate calculus example

Predicate logic employs variables for specific objects, function and relation symbols, and quantifiers (∀,∃). So, using our previous “Fido” example:

Fido is a dog	“P”	Df
Dogs like bones	“Q”	Bd
Fido likes bones	“R”	Bf

Denoting “any dog” by “d”, “is a dog” by “D” and “likes bones” by “B”:

$$\forall d \cdot (Df \wedge Bd) \rightarrow Bf \quad \text{or...}$$

$$\forall d{:}dog \cdot (Df \wedge Bd) \rightarrow Bf$$

Introducing Quantification

(making predicate calculus)

Quantification Places bounds on free variables (i.e. names of objects)

$P(x)$ *Is a unary predicate*

(1) $\exists x \cdot P(x)$ and (2) $\forall x \cdot P(x)$ *Produce propositions*

1. *There exists an object 'x' to which the predicate 'P(x)' applies.*
2. *For all objects 'x', the predicate 'P(x)' applies.*

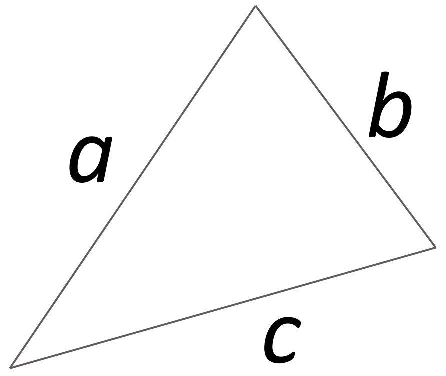
Some examples:

$\exists x{:}staff\_age \bullet x > 50$ meaning, there is staff who is older than 50;

$\forall x{:}names \bullet relatives(x)$ meaning, the persons by these names are all relatives.

Predicate Calculus in Definitions

- Can be viewed as conditional statements obeying propositional behaviour with specific values.
- **Consider a triangle** - We can say the following…



For any triangle:

1. It will consist of three sides;
2. Any of its sides will be greater than zero length;
3. The sum of the length of any two of its sides will be greater than the length of the remaining side.

Therefore…

Predicate Calculus Example

In predicate calculus form the basic properties of any triangle "T" could be written as follows** (using qualification and predicate notation)**:

T := if $1 \wedge 2 \wedge 3$, *or more formally…*

T := $\forall a,b,c \in \Re \bullet \mathcal{P}(a,b,c)$, *and more specifically…*

taking $\mathcal{P}(a,b,c) := ((a>0)\wedge(b>0)\wedge(c>0))\wedge((a+b>c)\wedge(b+c>a)\wedge(a+c>b))$

yields…

T := $\forall a,b,c \in \Re \bullet ((a>0)\wedge(b>0)\wedge(c>0))\wedge((a+b>c)\wedge(b+c>a)\wedge(a+c>b))$

A More Familiar Predicate Form

Consider the predicates:

numerically_bigger_than(x,y)

are_equal(a,b)

Can be written as…

$x > y$

$a = b$

Quantification Examples

$\exists\, i : 1..10 \cdot i^2 = 64$

$\exists\, proc : processors \cdot ProcessorState(proc, active)$

$\exists\, i : N \cdot i > 10 \;\; MonitorTemp = i$

$\exists\, m : AllocatedMonitors \cdot MonState(m, ready)$

$\exists\, i : 1..100; m : AllocatedMonitors \cdot activity(m, functioning) \wedge AmbientTemp = i$

$\exists\, r : CurrentReactors; m : AllocatedMonitors \cdot MonState(m, functioning) \wedge connected(r, m)$

More Examples *(loosely adopted from Behforooz, A.)*

$$\forall x,y,z \bullet x > y \wedge y > z \rightarrow x > z$$

$$\exists x \ni x > 10 \vee x + y < 100$$

$$\forall x,y \in N \rightarrow x + y \in N$$

$$\exists x,y \in \{1,2,3,4\} \ni x + y \in \{1,2,3,4\}$$

$$\forall x,y \in \{1,2,3,4\} \bullet x > y \rightarrow x - y \in \{1,2,3,4\}$$

$$\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$$

$$x > y \leftrightarrow x - y > 0$$

$$x + y > 0 \nrightarrow x > 0 \wedge y > 0$$

The

If x

is a

There exist x and y from the set {1,2,3,4} such that the sum of x and y is also a member of the set {1,2,3,4}

For all values x and y from the set

The complement (negation of) two

The numeric value x is greater than y if

n

If the sum of x and y is positive, it cannot be concluded that both x and y are positive

Algebraic Specifications

- A specification technique mainly used for abstract data types
- Based on a strong mathematical foundation – namely algebraic relationships and logical equivalence
- Have been in use for relatively long periods of time
- Are universal in their application
- Employ fundamental principles

Building Algebraic Specifications

- Clearly comprehend the system to model
- Determine the operations necessary for the system you have in mind
- Specify the relationship between the system’s operations
- Write the specification down according to adopted standard (most are based on the foundational concepts of the Common Algebraic Specification Language – CASL)

Algebraic Specification Schema Structure

Type: *<resulting type>*
Imports: *<what it uses/assumes>*

Signatures:
<a list of the abstract types that result from every relevant operation specified within the axiomatic part schema>

Axioms:
<a list of the relevant operations that will be used to prove situations based on logical equivalence>

Algebraic Specification Example

(1/2) (loosely adopted from Sommerville, I - 2009)

A list of elements:

Operations:

Action	***Name***	***Classification***
Create a new list	[Create]	*Basic*
Add an an element	[Add]	*Basic*
Get the first list element	[Get]	*Inspector*
Retain the list except the first element	[Tail]	*Extra*
Count length of list	[Count]	*Inspector*

Algebraic Specification Example

(2/2) (loosely adopted from Sommerville, I - 2009)

Type: *list*

Imports: *Z, Boolean*

Signatures:

Create() $\rightarrow$ list

Add(list, elem) $\rightarrow$ list

Get(list) $\rightarrow$ elem

Tail(list) $\rightarrow$ list

Count(list) $\rightarrow$ Z

Axiom construction:

Basics x ((Extras+B[a]) + (Inspectors+B[a]))
Therefore: 2x((1+0)+(2+0)) = 6 Axioms
Extras & Inspectors "act" on Basics

Axioms:

1) Count(Create()) = 0;

2) Count(Add(l, e)) = if Count(l) = 0 then 1 else count(l)+1;

3) Get(Create()) = error;

4) Get(Add(l, e)) = if count(l) = 0 then e else get(l);

5) Tail(Create()) = create();

6) Tail(Add(l, e)) = if count(l) = 0 then create() else l;

Algebraic Specification Example

(1/2) (loosely adopted from Pressman, R. S.)

A queue of integers:

Operations:

Action	***Name***	***Classification***
Add an item to end of queue	[Add]	*Basic*
Remove an item from end of queue	[Rem]	*Extra*
Check if queue empty	[IsEmpty]	*Inspector*
Read first queue item	[ReadFirst]	*Inspector*
Read last queue item	[RaadLast]	*Inspector*
Create a new queue	[Create]	*Basic*

Algebraic Specification Example

(2/2) (loosely adopted from Pressman, R. S.)

Type: *queue*

Imports: *Z, Boolean*

Signatures:

Create() $\rightarrow$ *queue*

Add(int, queue) $\rightarrow$ *queue*

Rem(int, queue) $\rightarrow$ *queue*

ReadFirst(queue) $\rightarrow$ *Z*

ReadLast(queue) $\rightarrow$ *Z*

IsEmpty(queue) $\rightarrow$ *Boolean*

Axiom construction:

Basics x ((Extras+B[a]) + (Inspectors+B[a]))
Therefore: 2x((1+0)+(3+0)) = 8 Axioms
Extras & Inspectors "act" on Basics

Axioms:

1) IsEmpty(Create()) = TRUE;

2) IsEmpty(Add(z, q)) = FALSE;

3) Rem(Create()) = error;

4) Rem(z, Add(z, q)) = q;

5) ReadFirst(Create()) = error;

6) ReadFirst(Add(z, q)) = if IsEmpty(q) then z
else ReadFirst(q);

7) ReadLast(Create()) = error

8) ReadLast(Add(z, q)) = if IsEmpty(q) then z
else ReadLast(q);

Algebraic Specification Example

(1/2) (dopted from Ghezzi, et al.)

String manipulation:

Operations:

Action	***Name***	***Classification***
Create a new string	[Create]	*Basic*
Concatenate strings	[Con]	*Extra*
Add a character	[Add]	*Basic*
Check string equality	[Equal]	*Inspector*
Count string length	[Count]	*Inspector*
Check for zero length	[isEmpty]	*Inspector*

Algebraic Specification Example

(2/2) (adopted from Ghezzi, et al.)

Type: *char, string*

Imports: *N, Boolean*

Signatures:

Create() → string

Con(str1, str2) → string

Add(char, str) → string

Equal(str1, str2) → Boolean

Count(str) → N

IsEmpty(str) → Boolean

Axiom construction:

Basics x ((Extras+B[a]) + (Inspectors+B[a]))

Therefore: 2x((1+0)+(1+2)) = 10 Axioms

Extras & Inspectors "act" on Basics

Axioms:

1) IsEmpty(Create()) = TRUE;

2) IsEmpty(Add(c, s)) = FALSE;

3) Count(Create()) = 0;

4) Count(Add(c, s) = if count(s) = 0 then 1 else Count(s)+1;

5) Con(s, Create()) = s;

6) Con(s_1, Add(c, s_2)) = Add(c, con(s_1, s_2));

7) Equal(Create(),Create()) = TRUE;

8) Equal(Create(), Add(c, s)) = FALSE;

9) Equal(Add(c, s_1), Add(c, s_2)) = Equal(s_1, s_2)

10) Equal(Add(c, s), create()) = FALSE;

Other Examples

Various other examples may be discussed during lectures using other documents.

The Z-Specification Language

- Attempts to place a notational framework on formal system specification
- Based on set theory
- Is model-based (relies on well understood mathematical entities and their relationship)
- Equally used to model (specify) state as well as operations on states

Some Basic Z-Schema Examples *(1/2)*

$a : N$
$b : \{7, 1, 3, 24\}$

$a \in b$

a is a natural number and b is a set formed of natural numbers as shown. a is contained in b.

Note: Generically, to indicate "a set of", the notation "P (with a hollow stem)".
Example "b: PN"
Linear equivalent would be: [a:N; b:$\{7,1,3,24\}$ | $a \in b$]

Some Basic Z-Schema Examples *(2/2)*

$$\begin{array}{|l}\hline a, b : N \\ c : PN \\ \hline a \in c \\ b \in c \\ \hline\end{array}$$

$$\begin{array}{|l}\hline a, b : N \\ c : PN \\ \hline a \in c \wedge b \in c \\ \hline\end{array}$$

$$a, b : N; c : PN \mid a \in c \wedge b \in c$$

Naming Schemas

MonCondition

$$MonNo : N$$
$$AvailableMonitors : PN$$

$$MonNo \in AvailableMonitors$$

Or…

$$MonCondition \stackrel{\text{def}}{=}$$
$$[MonNo : N;\ AvailableMonitors : PN \mid MonNo \in AvailableMonitors]$$

Z-Schema Conventions

- Delta
  - Denoted by the Greek literal (Δ)
  - Used to extend the schema components to indicate update operations, i.e. changes in state variables (updating operations).
- "Xi"
  - Denoted by the Greek literal (Ξ)
  - Used to indicate that stored data is not affected, i.e. enquiry operations.

Delta Convention Examples *(taken from Ince)*

Specify a system which will keep track of students who have handed in Assignments. There are clearly three sets involved…

Class	*(all the students in the class)*
HandedIn	*(all the students in the class who have handed in their assignment)*
NotHandedIn	*(all the students in the class who have not handed in their assignment)*

$\Delta Assignment$

$$Class, HandedIn, NotHandedIn : \mathbb{P}\ STUDENTS$$
$$Class', HandedIn', NotHandedIn' : \mathbb{P}\ STUDENTS$$

$$HandedIn \cup NotHandedIn = Class$$
$$HandedIn \cap NotHandedIn = \emptyset$$
$$HandedIn' \cup NotHandedIn' = Class'$$
$$HandedIn' \cap NotHandedIn' = \emptyset$$

Use of a Delta Schema (based on previous example)

Model the handing in of a student assignment:

HandIn

$stud? : STUDENTS$

$\Delta\, Assignment$

$Stud? \in NotHandedIn$
$NotHandedIn' = NotHandedIn \setminus \{Stud?\}$
$HandedIn' = HandedIn \cup \{Stud?\}$
$Class' = Class$

Use of a "Xi" Schema (based on previous example)

Model a query for the number of students who have handed in:

$$\Xi\ Assignment$$

$$Class, HandedIn, NotHandedIn : \mathbb{P}\ STUDENTS$$
$$Class', HandedIn', NotHandedIn' : \mathbb{P}\ STUDENTS$$

$$NotHandedIn' = NotHandedIn$$
$$HandedIn' = HandedIn$$
$$Class' = Class$$

Therefore:

$$AssignQuery \overset{\text{def}}{=}$$
$$[HandedIn! : \mathbb{N};\ \Xi\ Assignment \mid HandedIn! = \#HandedIn]$$

Schema Inclusion

A simple example of this will be presented during lectures.

Another Schema Inclusion Example *(1/2)*

FileStatus

$$
\begin{array}{l}
AllFiles, FreeFile, FilesInUse : \mathbb{P}\ FILES \\
File : FILES \\
RegisteredUsers : \mathbb{P}\ NAMES \\
User : NAMES \\
\hline
User \in RegisteredUsers \\
File \in AllFiles \\
AllFiles = FreeFiles \cup FilesInUse \\
FreeFile = AllFiles \setminus FilesInUse \\
FreeFiles \cap FilesInUse = \varnothing \\
FreeFile \notin FilesInUse
\end{array}
$$

UserStatus

$$
\begin{array}{l}
FileStatus \\
InvalidUsers : \mathbb{P}\ NAMES \\
\hline
User \in RegisteredUsers \\
User \notin InvalidUsers \\
RegisteredUsers \cap InvalidUsers = \varnothing
\end{array}
$$

Another Schema Inclusion Example *(2/2)*

Results in the following schema…

FileAndUserStatus

$$AllFiles, FreeFile, FilesInUse : \mathbb{P}\ FILES$$
$$File : FILES$$
$$RegisteredUser, InvalidUser : \mathbb{P}\ NAMES$$
$$User : NAMES$$

$$User \in RegisteredUsers$$
$$User \notin InvalidUsers$$
$$RegisteredUsers \cap InvalidUsers = \emptyset$$
$$File \in AllFiles$$
$$AllFiles = FreeFiles \cup FilesInUse$$
$$FreeFile = AllFiles \setminus FilesInUse$$
$$FreeFiles \cap FilesInUse = \emptyset$$
$$FreeFile \notin FilesInUse$$

Another Schema Inclusion Example *(taken from Ince)*

```
┌─ SetInv ─────────────────────────────
│ Upper, lower : PN
│ MaxSize: N
├──────────────
│ #upper + #lower ≤ MaxSize
└──────────────────────────────────────
```

$$\#upper + \#lower \leq MaxSize$$

```
┌─ MidInv ─────────────────────────────
│ middle: PN
│ SetInv
├──────────────
│ middle ⊂ upper ∪ lower
└──────────────────────────────────────
```

$$middle \subset upper \cup lower$$

The above schemas result in…

```
┌─ MidInv ─────────────────────────────
│ Middle : PN
│ Upper, lower: PN
│ MaxSize : N
├──────────────
│ middle ⊂ upper ∪ lower
│ #upper + #lower ≤ MaxSize
└──────────────────────────────────────
```

$$middle \subset upper \cup lower$$
$$\#upper + \#lower \leq MaxSize$$

Sequences

- *Are not* sets
- Can be viewed as collections with predefined constraints
- Exist in different forms
- Can have operations applied to them
- Widespread use in computer systems

Types of Sequences

- Normal (including empty)

seq

- Non-empty

seq_1

- Injective (not containing duplicates)

iseq

Therefore…

Formal Sequence Definitions

- Definitions…

  seq *T* = = { *f* : N ⍰ *T* | dom *f* = 1 .. #*f* }

  seq$_1$ *T* = = { *f* : seq *T* | #*f* > 0 }

  iseq *T* = = seq *T* ∩ (N ⍰ *T*)

- Some examples…

  E.g. of (seq N) is {1 ⍰ 3, 2 ⍰ 9, 3 ⍰ 9, 4 ⍰ 11}

  written as ⟨3,9,9,11⟩

  E.g. of (iseq *files*) is {1 ⍰ *UpdateFile*, 2 ⍰ *LogFile*, 3 ⍰ *TaxFile*}

  written as ⟨*UpdateFile*,*LogFile*,*TaxFile*⟩

Sequences in Z

Examples of these will be presented during lectures.

Sequence (in Z) Example

FileQueue

$InQueue, OutQueue : \text{seq } Files$

$\#InQueue < \#OutQueue$

Rentals

$Pending, Overdue : \text{seq } ID$

$MostOverdue!, SoonToBeOverdue! : ID$

$MostOverdue! = \text{head } Overdue$

$SoonToBeOverdue! = \text{head } Pending$

Heads n' Tails

- Formally defining the “head”, “last”, “tail”, and “front” sequence operators using a Z-schema.

$$
\begin{array}{|l}
\textit{[SeqOps]} \\
\hline
head, last : seq_1\ SeqOps \rightarrow SeqOps \\
tail, front : seq_1\ SeqOps \rightarrow seq\ SeqOps \\
\hline
\forall s : seq_1\ SeqOps \bullet \\
\qquad head\ s = s(1) \wedge \\
\qquad last\ s = s(\#s) \wedge \\
\qquad tail\ s = (\lambda\ n : 1\ ..\ \#s\text{-}1 \bullet s(n+1)) \boxed{?} (\{1\} \boxed{?} s) \wedge \\
\qquad front\ s = (n : 1\ ..\ \#s\text{-}1 \bullet s(n)) \\
\hline
\end{array}
$$

Summary

- Formal approaches
- Sets, propositions and predicates
- Algebraic specifications
- Z-Schemas