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

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

Ernest Cachia - Department of Computer Information Systems
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-compressible 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”

Generic introduction to formal specification

- 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];

A. Hall. “Seven myths of formal methods”. In: Software, IEEE 7.5 (Sept. 1990), pp. 11–19;
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.

Position of Formal methods in SE

Formalism in Software Engineering

- Define requirements
- Implementation process

- Specify requirements
- Model functionality

- Formal specification
- Architectural design

- Detailed design
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: \( \{\text{car, bus, truck, motorbike}\} \subseteq \{\text{vehicles}\} \)

• Propositional calculus (Zeroth-order logic)
Reasoning about statements and what can be inferred from them
  e.g. \( p = \text{A lab is a room}; \ q = \text{AC is off in an empty room}; \ r = \text{AC is off.} \)
  Formally expressed as: \( p \land q \Rightarrow r \)

• 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 Ø
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\}  
Kiev ∉ \{Maltese towns\}  
#\{joe, veronica, mark\} = 3  
\{paul, richard, claire, george\} ⊆ \{Group B\}  
P\{Dynamo, CSKA\} = \{ {}, {Dynamo}, {CSKA}, {Dynamo, CSKA} \}  
\{ford, toyota, kia\} ∪ \{toyota, audi\} = \{ford, toyota, kia, audi\}  
\{ford, toyota, mercedes\} ∩ \{ford, audi\} = \{ford\}  
\{ford, toyota, subaru\} \setminus \{ford, toyota, kia\} = \{subaru\}  
\{ford, toyota, subaru\} Δ \{ford, toyota, kia\} = \{subaru, kia\}  
\{1, 2, 3\} × \{2, 4\} = \{ {1,2}, {2,2}, {3,2}, {1,4}, {2,4}, {3,4} \}
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 | condition • operation}
```

**In general:**

```
{Signature | Predicate • Term}
```

**...or in formal notation:**

```
{x : X | P(x) • E(x)}
```
Set Construction Examples

Alternate even numbers:

\[ \{ x : N \mid x \mod 2 = 0 \cdot 2 \ast x \} = \{0,4,8,12,16,20,...\} \]

Tens:

\[ \{ x : N \mid x \cdot 10 \ast x \} = \{0,10,20,30,40,...\} \]

Squares of multiples of 4 (excluding zero):

\[ \{ x : Z \mid (x \mod 4 = 0) \land (x > 0) \cdot x \ast x \} = \{16,64,144,256,...\} \]
Some Set-Based Specification Exercises

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

\{n : \mathbb{N} \mid n > 10 \land n < 20 \cdot n\} \quad \{11, 12, 13, \ldots\}
\{n : \mathbb{N} \mid n^3 > 10 \cdot n\} \quad \{3, 4, 5, \ldots\}
\{x, y : \mathbb{N} \mid x + y = 100 \cdot (x, y)\} \quad \{(0,100), (1,99), (2,98), \ldots\}
\{x, y : \mathbb{N} \mid x + y = 5 \cdot x^2 + y^2\} \quad \{25, 17, 13, \ldots\}

Interpret the following:

\{m : \text{monitors} \mid \text{MonitorState}(m, \text{on}) \cdot m\} \quad \text{Monitors that are on}
\{f : \text{SysFiles} \mid f \in \text{DelFiles} \land f \in \text{ArcFiles} \cdot f\} \quad \text{Deleted system archive files}

Write the comprehensive specification of:

\{(10,100), (11,121), (12,144), (13,169), (14,196)\} \quad \{x : \mathbb{N} \mid x \geq 10 \cdot (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

- Sit down.
- How are you today?
- Get my tea, please.
- What is the weather like?
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): $\neg$
- Conjunction (and): $\land$
- Disjunction (or): $\lor$
- Material implication (if...then): $\rightarrow$
- Biconditional (if and only if): $\leftrightarrow$

Some symbolic examples:

$\neg ((P \lor Q) \rightarrow Q)$ true
$\neg ((P \land Q) \lor \neg R) \leftrightarrow P$ false
$\neg P \land (P \lor (Q \rightarrow P))$ false

$((P \rightarrow Q) \land (R \rightarrow S) \land (P \lor R)) \rightarrow (Q \lor S)$ 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:*

\[ a \land \neg a \]  contradiction

\[ a \lor \neg a \]  tautology

\[ (a \land b \land c) \Rightarrow (c \Rightarrow a) \]  tautology
De Morgan’s Laws (at the heart of transformation)

\[ \neg (P \land Q) \iff \neg P \lor \neg Q \]
\[ \neg (P \lor Q) \iff \neg P \land \neg Q \]

Some other propositional calculus transformation rules regularly used:

- **Implication (IF)**
  \[ (P \rightarrow Q) \land (Q \rightarrow R) \rightarrow (P \rightarrow R) \]

- **Bicondition (IF AND ONLY IF)**
  \[ (P \rightarrow Q) \land (Q \rightarrow P) \rightarrow (P \iff Q) \]

- **Conjunction**
  \[ (P \land Q) \]

- **Disjunction introduction**
  \[ P \rightarrow (P \lor Q) \]

- **Disjunctive syllogism**
  \[ ((P \lor Q) \land \neg P) \rightarrow Q \]

- **Constructive**
  \[ (((P \rightarrow Q) \land (R \rightarrow S)) \land (P \lor R)) \rightarrow (Q \lor S) \]

- **Absorption**
  \[ (P \rightarrow Q) \iff (P \rightarrow (P \land Q)) \]

- **Hypothetical syllogism**
  \[ ((P \rightarrow Q) \land (Q \rightarrow R)) \rightarrow (P \rightarrow R) \]
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.”

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.

\[ \text{alert} \leftrightarrow \text{armed} \land \text{practice} \]

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

\[ \text{teaching} \land \text{practice} \rightarrow \text{alert} \]

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

\[ \text{teaching} \land \text{practice} \land \neg \text{alert} \]

Therefore...
Detecting Contradictions

From the previous analysed specification:
alert ↔ armed ∧ practice
teaching ∧ practice → alert
teaching ∧ practice ∧ ¬alert

The second and third propositions yield a contradiction:
teaching ∧ practice → alert
teaching ∧ practice ∧ ¬alert
alert ∧ ¬alert ...contradiction!

Furthermore, the first proposition yields another contradiction:
teaching ∧ practice ∧ ¬(armed ∧ practice) simplifies to...
alert ∧ ¬armed ∨ ¬practice ...contradiction because being alert requires being armed!
“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:
  
  - $\text{dog}(\text{fido}) = \text{true}; \quad \text{Df}$
  - $\text{dog}(\text{lecturer}) = \text{false}. \quad \text{Dl} \quad \cdots(?)$

- Examples of $n$-ary predicates:
  
  - $\text{owned}(\text{Labrador}, \text{Boxer}); \quad \text{O}(l,b)$
  - $\text{father}(\text{John}, \text{Mary}); \quad \text{F}(j,m)$
  - $\text{team}(\text{Paul}, \text{Albert}, \text{Vincent}). \quad \text{T}(p,a,v)$
Introducing Quantification  
(making predicate calculus)

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

\[ P(x) \text{ is a unary predicate} \]

\[ \exists x \cdot P(x) \text{ and } \forall x \cdot P(x) \text{ Produce propositions} \]

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:
\[ \exists x:staff\_age \cdot x > 50 \] meaning, there is staff who is older than 50;
\[ \forall x:names \cdot relatives(x) \] meaning, the persons by these names are all relatives.
Predicate logic employs variables for specific objects, function and relation symbols, and quantifiers (\(\forall, \exists\)). So, using our previous “Fido” example:

Fido is a dog \(\text{“P”} \quad Df\)
Dogs like bones \(\text{“Q”} \quad Bd\)
Fido likes bones \(\text{“R”} \quad Bf\)

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

\[ \forall d \cdot (Df \land Bd) \rightarrow Bf \quad \text{or...} \]
\[ \forall d:\text{dog} \cdot (Df \land Bd) \rightarrow Bf \]
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...
In predicate calculus form the basic properties of any triangle “T” could be written as follows (using qualification and predicate notation):

\[ T := \text{if } 1 \land 2 \land 3, \quad \text{or more formally...} \]
\[ T := \forall a, b, c \in \mathbb{R} \bullet P(a, b, c), \quad \text{and more specifically...} \]

taking \( P(a, b, c) := ((a > 0) \land (b > 0) \land (c > 0)) \land ((a + b > c) \land (b + c > a) \land (a + c > b)) \)

yields...
\[ T := \forall a, b, c \in \mathbb{R} \bullet ((a > 0) \land (b > 0) \land (c > 0)) \land ((a + b > c) \land (b + c > a) \land (a + c > b)) \]
Consider the predicates:

\[
\text{numerically\_bigger\_than}(x, y) \\
\text{are\_equal}(a, b)
\]

Can be written as...

\[
x > y \\
a = b
\]
\[ \exists i : 1..10 \cdot i^2 = 64 \]

\[ \exists \text{proc} : \text{processors} \cdot \text{ProcessorState}(\text{proc}, \text{active}) \]

\[ \exists i : \text{N} \cdot i > 10 \land \text{MonitorTemp} = i \]

\[ \exists m : \text{AllocatedMonitors} \cdot \text{MonState}(m, \text{ready}) \]

\[ \exists i : 1..100; m : \text{AllocatedMonitors} \cdot \text{activity}(m, \text{functioning}) \land \text{AmbientTemp} = i \]

\[ \exists r : \text{CurrentReactors}; m : \text{AllocatedMonitors} \cdot \text{MonState}(m, \text{functioning}) \land \text{connected}(r, m) \]
More Examples (loosely adopted from Behforooz, A.)

\[ \forall x, y, z \cdot x > y \land y > z \rightarrow x > z \]

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\}

\[ \exists x \exists x > 10 \lor x + y < 100 \]

If x is a numeric value, either x is greater than 10 or for some value y the sum of x and y is less than 100

\[ \forall x, y \in N \rightarrow x + y \in N \]

If x and y are natural numbers, then x+y is also a natural number

\[ \exists x, y \in \{1,2,3,4\} \exists x + y \in \{1,2,3,4\} \]

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\}

\[ \forall x, y \in \{1,2,3,4\} \cdot x > y \rightarrow x - y \in \{1,2,3,4\} \]

For all values x and y from the set \{1,2,3,4\} for which x is greater than y, the difference between x and y is also an element of the set \{1,2,3,4\}

\[ \neg (p \land q) \leftrightarrow \neg p \lor \neg q \]

The complement (negation of) two logical AND-ed values is the same as the complement of each OR-ed value

\[ x > y \leftrightarrow x - y > 0 \]

The numeric value x is greater than y if and only if the difference between x and y is positive

\[ x + y > 0 \leftrightarrow x > 0 \land y > 0 \]

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

<table>
<thead>
<tr>
<th>Type:</th>
<th>(&lt;\text{resulting type}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imports:</td>
<td>(&lt;\text{what it uses/assumes}&gt;)</td>
</tr>
<tr>
<td>Signatures:</td>
<td>(&lt;\text{a list of the abstract types that result from every relevant operation specified within the axiomatic part schema}&gt;)</td>
</tr>
<tr>
<td>Axioms:</td>
<td>(&lt;\text{a list of the relevant operations that will be used to prove situations based on logical equivalence}&gt;)</td>
</tr>
</tbody>
</table>
A list of elements:

**Operations:**

<table>
<thead>
<tr>
<th>Action</th>
<th>Name</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new list</td>
<td>[Create]</td>
<td>Basic</td>
</tr>
<tr>
<td>Add an element</td>
<td>[Add]</td>
<td>Basic</td>
</tr>
<tr>
<td>Get the first list element</td>
<td>[Get]</td>
<td>Inspector</td>
</tr>
<tr>
<td>Retain the list except the first</td>
<td>[Tail]</td>
<td>Extra</td>
</tr>
<tr>
<td>element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count length of list</td>
<td>[Count]</td>
<td>Inspector</td>
</tr>
</tbody>
</table>
**Type:** list  
**Imports:** Z, Boolean

**Signatures:**  
Create() → list  
Add(list, elem) → list  
Get(list) → elem  
Tail(list) → list  
Count(list) → 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;
A queue of integers:

**Operations:**

<table>
<thead>
<tr>
<th><strong>Action</strong></th>
<th><strong>Name</strong></th>
<th><strong>Classification</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Add an item to end of queue</td>
<td>[Add]</td>
<td>Basic</td>
</tr>
<tr>
<td>Remove an item from end of queue</td>
<td>[Rem]</td>
<td>Extra</td>
</tr>
<tr>
<td>Check if queue empty</td>
<td>[IsEmpty]</td>
<td>Inspector</td>
</tr>
<tr>
<td>Read first queue item</td>
<td>[ReadFirst]</td>
<td>Inspector</td>
</tr>
<tr>
<td>Read last queue item</td>
<td>[RaadLast]</td>
<td>Inspector</td>
</tr>
<tr>
<td>Create a new queue</td>
<td>[Create]</td>
<td>Basic</td>
</tr>
</tbody>
</table>
**Type:** queue

**Imports:** Z, Boolean

**Signatures:**

Create() → queue
Add(int, queue) → queue
Rem(int, queue) → queue
ReadFirst(queue) → Z
ReadLast(queue) → Z
IsEmpty(queue) → Boolean

**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);

**Axiom construction:**

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

Ernest Cachia - Department of Computer Information Systems
### Algebraic Specification Example

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

**String manipulation:**

**Operations:**

<table>
<thead>
<tr>
<th>Action</th>
<th>Name</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new string</td>
<td>[Create]</td>
<td>Basic</td>
</tr>
<tr>
<td>Concatenate strings</td>
<td>[Con]</td>
<td>Extra</td>
</tr>
<tr>
<td>Add a character</td>
<td>[Add]</td>
<td>Basic</td>
</tr>
<tr>
<td>Check string equality</td>
<td>[Equal]</td>
<td>Inspector</td>
</tr>
<tr>
<td>Count string length</td>
<td>[Count]</td>
<td>Inspector</td>
</tr>
<tr>
<td>Check for zero length</td>
<td>[isEmpty]</td>
<td>Inspector</td>
</tr>
</tbody>
</table>
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

**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;

**Axiom construction:**
Basics x ((Extras+B[a]) + (Inspectors+B[a]))  
Therefore: 2x((1+0)+(1+2)) = 10 Axioms  
Extras & Inspectors “act” on Basics
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)

\[
a : N \\
b : \{7, 1, 3, 24\} \\
a \in b
\]

\textit{a is a natural number and b is a set formed of natural numbers as shown. a is contained in b.}

\textit{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)

\[
\begin{align*}
\text{if } a, b : N \\
c : PN \\
\quad a \in c \\
b \in c
\end{align*}
\]

Is equivalent to…

Or linearly…

\[
\begin{align*}
\text{if } a, b : N; c : PN \mid a \in c \land b \in c
\end{align*}
\]
Naming Schemas

\[ Mon\text{Condition} \]

\[ MonNo : N \]
\[ Available\text{Monitors} : PN \]

\[ MonNo \in Available\text{Monitors} \]

Or...

\[ Mon\text{Condition} \overset{\text{def}}{=} [MonNo : N; Available\text{Monitors} : PN | MonNo \in Available\text{Monitors}] \]
**Z-Schema Conventions**

- **Delta**
  - Denoted by the Greek literal ($\Delta$)
  - Used to extend the schema components to indicate update operations, i.e. changes in state variables (updating operations).

- **“Xi”**
  - Denoted by the Greek literal ($\Xi$)
  - Used to indicate that stored data is not affected, i.e. enquiry operations.
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)

\[ \Delta \text{Assignment} \]

\[
\begin{align*}
\text{Class, HandedIn, NotHandedIn} : & \ P \ \text{STUDENTS} \\
\text{Class', HandedIn', NotHandedIn'} : & \ P \ \text{STUDENTS} \\
\text{HandedIn} \cup \text{NotHandedIn} = & \ \text{Class} \\
\text{HandedIn} \cap \text{NotHandedIn} = & \ \emptyset \\
\text{HandedIn'} \cup \text{NotHandedIn'} = & \ \text{Class'} \\
\text{HandedIn'} \cap \text{NotHandedIn'} = & \ \emptyset
\end{align*}
\]
Model the handing in of a student assignment:

\[
\text{HandIn}
\]

\[
\begin{align*}
\text{stud} & \in \text{STUDENTS} \\
\Delta & \text{Assignment} \\
\text{Stud} & \in \text{NotHandedIn} \\
\text{NotHandedIn} & = \text{NotHandedIn} \setminus \{\text{Stud} \} \\
\text{HandedIn} & = \text{HandedIn} \cup \{\text{Stud} \} \\
\text{Class} & = \text{Class}
\end{align*}
\]
Model a query for the number of students who have handed in:

\[ \exists \text{Assignment} \]

\[ \text{Class}, \text{HandedIn}, \text{NotHandedIn} \colon \text{P STUDENTS} \]
\[ \text{Class}', \text{HandedIn}', \text{NotHandedIn}' \colon \text{P STUDENTS} \]

\[ \text{NotHandedIn}' = \text{NotHandedIn} \]
\[ \text{HandedIn}' = \text{HandedIn} \]
\[ \text{Class}' = \text{Class} \]

Therefore:

\[ \text{AssignQuery} \overset{\text{def}}{=} [\text{HandedIn}' \colon \text{N}; \exists \text{Assignment} \mid \text{HandedIn}' = \#\text{HandedIn}] \]
A simple example of this will be presented during lectures.
Another Schema Inclusion Example (1/2)

FileStatus

AllFiles, FreeFile, FilesInUse : P FILES
File : FILES
RegisteredUsers : P NAMES
User : NAMES

User ∈ RegisteredUsers
File ∈ AllFiles
AllFiles = FreeFiles ∪ FilesInUse
FreeFile = AllFiles \ FilesInUse
FreeFiles ∩ FilesInUse = ∅
FreeFile ̸∈ FilesInUse

UserStatus

FileStatus
InvalidUsers : P NAMES

User ∈ RegisteredUsers
User ̸∈ InvalidUsers
RegisteredUsers ∩ InvalidUsers = ∅
Results in the following schema...

**FileAndUserStatus**

- AllFiles, FreeFile, FilesInUse: $P$ FILES
- File: FILES
- RegisteredUser, InvalidUser: $P$ 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)

\[
\text{SetInv} \\
\text{Upper, lower : PN} \\
\text{MaxSize: N} \\
\#upper + \#lower \leq \text{MaxSize}
\]

\[
\text{MidInv} \\
\text{middle: PN} \\
\text{SetInv} \\
\text{middle } \subseteq \text{upper } \cup \text{lower}
\]

The above schemas result in...

\[
\text{MidInv} \\
\text{Middle : PN} \\
\text{Upper, lower: PN} \\
\text{MaxSize : N} \\
\text{middle } \subseteq \text{upper } \cup \text{lower} \\
\#upper + \#lower \leq \text{MaxSize}
\]
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...

Ernest Cachia - Department of Computer Information Systems
Formal Sequence Definitions

• Definitions…

\[ \text{seq } T = = \{ f : \mathbb{N} \rightarrow T \mid \text{dom } f = 1 \ldots #f \} \]
\[ \text{seq}_{1} T = = \{ f : \text{seq } T \mid #f > 0 \} \]
\[ \text{iseq } T = = \text{seq } T \cap (\mathbb{N} \rightarrow T) \]

• Some examples…

E.g. of (seq N) is \{1 \rightarrow 3, 2 \rightarrow 9, 3 \rightarrow 9, 4 \rightarrow 11\}
written as \langle 3,9,9,11 \rangle

E.g. of (iseq files) is \{1 \rightarrow UpdateFile, 2 \rightarrow LogFile, 3 \rightarrow TaxFile\}
written as \langle UpdateFile,LogFile,TaxFile \rangle
Sequences in $\mathbb{Z}$

Examples of these will be presented during lectures.
**FileQueue**

\[ \text{InQueue, OutQueue : seq Files} \]

\[ \#\text{InQueue} < \#\text{OutQueue} \]

**Rentals**

\[ \text{Pending, Overdue : seq ID} \]

\[ \text{MostOverdue!, SoonToBeOverdue! : ID} \]

\[ \text{MostOverdue!} = \text{head Overdue} \]

\[ \text{SoonToBeOverdue!} = \text{head Pending} \]
Heads n’ Tails

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

\[ \text{SeqOps} \]

\[
\begin{align*}
\text{head}, \text{last} & : \text{seq} \implies \text{SeqOps} \\
\text{tail}, \text{front} & : \text{seq} \implies \text{seq} \implies \text{SeqOps} \\
\forall s : \text{seq} \implies \text{SeqOps} \implies \\
\text{head s} & = s(1) \land \\
\text{last s} & = s(#s) \land \\
\text{tail s} & = (\lambda n : 1 .. #s-1 \bullet s(n+1)) \square (\{1\} \square s) \land \\
\text{front s} & = (n : 1 .. #s-1 \bullet s(n))
\end{align*}
\]
Summary

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