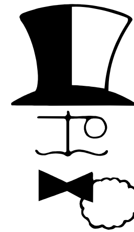
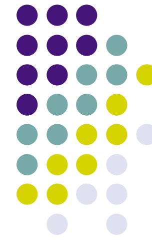
Formal Specifications





Software Specifications

- Architecture design is the activity of partitioning the requirements to software subsystems and is an essential prerequisite for specification.
- It provides the logical elements to specify.

Formal Specifications

- A formal specification is a specification expressed in a language whose vocabulary, syntax and semantics are formally defined.
- It cannot be based on natural language but instead must be based on **mathematics**.

Advantages

- The development of a formal specification provides insights into and understanding of the software requirements and the software design.
- Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specification.

Advantages

- Formal specifications have the possibility of
  - automated processing,
  - animation of a specification to provide a prototype system, and
  - mathematical study.
- Formal specs may be used as a guide to the creation of test cases for any particular component of the system.

Disadvantages

- Software management is inherently conservative and is unwilling to adopt new techniques if the payoff is not immediately obvious.
- Most analysts and programmers have not been trained in formal specification techniques.

Disadvantages

- Clients are not likely to be familiar with formal specifications and may be unwilling to fund development activities that they do not understand and therefore cannot control.
- Some classes of systems are still difficult to specify using current techniques.

Disadvantages

- There is widespread ignorance of current specification techniques and their uses.
- Most research efforts have been directed at the development of notations and not with tool support.

Notations

- Notations such as VDM involve the use of a number of specialized symbols which must be memorized and this leads to a significant learning curve.
- Some notations incorporate a mnemonic notation that is less alien and can be typed in using a standard keyboard.

Notations

- The specification language, Z (pronounced 'zed') uses graphics to structure specifications.
- This improves its readability and encourages incremental development of specifications.

Developing a Simple Formal Specification

- The simplest form is axiomatic specification where a system is represented as
  - a set of functions (which are stateless) and
  - each function is specified using pre- and post-conditions.
- This technique is currently used only for small systems or system components.

Pre- and Post-Conditions

- These conditions are predicates over the inputs and outputs of a function.
- A predicate is simply a boolean expression which is true or false and whose variables are the parameters of the function being specified.

Pre- and Post-Conditions

- Predicates include
  - operators (such as =, >, <, not, and, or),
  - the universal and existential quantifiers, and
  - the operator in which is used to select the range over which the quantifier applies.

The Development of an Axiomatic Spec of a Function

- Establish the range of the input parameters over which the function is intended to behave correctly. Specify the input parameter constraints as a predicate.
- Specify a predicate defining a condition which must hold on the output of the function if it behaves correctly.

The Development of an Axiomatic Spec of a Function

- Establish what changes (if any) are made to the function's input parameters and specify these.
- Combine these into pre- and post-conditions for the function.

An Example: Search

function Search (X:INTEGER_ARRAY;
  Key:INTEGER) return INTEGER;

Pre: $\exists$ i $\in$ X'FIRST..X'LAST:X(i) = Key

Post: X''(Search(X,Key)) = Key and X = X''

- The input array is unchanged by the search.
- Returns the value of the index of the element which is equal to the key.

Search of an Ordered Array

- The specification now includes an additional clause in the pre-condition.

function Search (X:INTEGER_ARRAY;
Key:INTEGER) return INTEGER;

Pre: $\exists$ i $\in$ X'FIRST..X'LAST: X(i) = Key $\wedge$

$\forall$ i,j $\in$ X'FIRST..X'LAST: i < j $\Rightarrow$ X(i) <= X(j)

Post: X''(Search(X,Key)) = Key and X = X''

The Use of an Error Predicate

- Specifications should also set out the behaviour of a component if it is presented with unexpected input.
- One approach is to have a number of pre/post-condition pairs depending on the number of erroneous input ranges.

The Use of an Error Predicate

function Search (X:INTEGER_ARRAY;
Key:INTEGER) return INTEGER;

Pre: $\exists$ i $\in$ X'FIRST..X'LAST: X(i) = Key

Post: X''(Search(X,Key)) = Key $\wedge$ X = X''

Error: Search(X,Key) = X'LAST + 1

The Use of Structure

- Large specifications are hard to understand so it is important that a specification language contains structuring facilities which allow specifications to be developed incrementally.
  - In our example, if we wanted the function to take a number of arrays as input parameters, we could name and parameterize a predicate.

The Use of Structure

Ordered (X:INTEGER_ARRAY) =
$\forall$ i,j $\in$ X'FIRST..X'LAST:i < j $\Rightarrow$
X(i) <= X(j)

Summary

- Considerations involved in creating a formal specification include
  - the conditions under which the software component behaves as anticipated,
  - erroneous input conditions,
  - the outputs of components when presented with erroneous input,
  - the input transformations,
  - the effect on the input parameters of a component.

Theoretical Aspects: Formally

- A formal specification language is a triple:

  < Syn, Sem, Sat >

  - where **Syn** and **Sem** are sets and **Sat**, a subset of **Syn X Sem**, is a relation between them.
  - **Syn** is called the language's syntactic domain, **Sem** is the semantic domain and **Sat** is the satisfies relation.

Theoretical Aspects: Formally

- Given a specification language,

  <Syn, Sem, Sat>

- if **Sat(syn,sem)** then **syn** is a *specification* of **sem** and **sem** is a *specificand* of **syn**.

Less Formally!

- A formal specification language provides
  - a notation (its syntactic domain),
  - a universe of objects (its semantic domain), and
  - a precise rule defining which objects satisfy each specification.

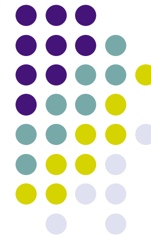
Less Formally!

- A specification is a sentence written in terms of the elements of the syntactic domain; it denotes a specificand set, a subset of the semantic domain.
- A specificand is an object satisfying a specification -- the satisfies relation provides the meaning for the syntactic elements.

An Example of a Simple Specification Language

- Backus-Naur form:
  - syntactic domain ⇒ a set of grammars
  - semantic domain ⇒ a set of strings
  - Every string is a specificand of each grammar that generates it.
  - Every specificand set is a formal language.

A Pragmatic Approach



Users

- Besides specification writers, there are several kinds of specification reader: customers, implementers, clients, verifiers, and even machine tools.
- Some languages may be more suitable to one type of specification user than to another.

Users

- The appropriate domain of applicability and target readers should be part of any language description.

Uses

- Formal methods can be applied in all phases of system development.
- The greatest benefit often comes from the process of formalizing rather than from the end result.

Requirements Analysis

- Helps clarify a customer's set of informally stated requirements.
- Reveals contradictions, ambiguities, and incompleteness.
- A specifier has a better chance of asking pertinent questions and evaluating customer responses.

Systems Design

- Two of the most important activities are decomposition and refinement.
- VDM, Z, and Larch are formal methods that are especially suitable for system design.
- Decomposition is the process of partitioning a system into smaller modules.

Systems Design

- Specifiers can write specs to capture precisely the interfaces between these modules.
- The interface provides a place for recording design decisions.

Systems Design

- Refinement involves working at different levels of abstraction -- refining a single module at one level to be a collection of modules at a lower level.
- Each refinement step requires showing that a specification at one level satisfies a higher level specification.

Systems Design

- Proving satisfaction often generates additional assumptions, called proof obligations, that must be discharged for the proof to be valid.
  - A formal method provides the language to state these proof obligations precisely and the framework to carry out the proof.

System Verification

- Verification is the process of showing that a system satisfies its specification.
- Formal verification is impossible without a formal specification.

System Verification

- You may not be able to verify an entire system but you can verify smaller, critical pieces.
  - But it is hard to explicitly state the assumptions about the environment in which each critical piece is placed.

System Validation

- Formal methods can aid in testing and debugging.
- They can be used to generate test cases for black-box testing.
- Specs that explicitly state assumptions on a module's use identify test cases for boundary conditions.

System Documentation

- A spec is a description alternative to system implementation.
- Primary intended use is to capture the “what” in “what does the system do?” rather than the “how”.

System Analysis and Evaluation

- To learn from the experience of building a system, developers should do a critical analysis of its functionality and performance once it has been built and tested.
- The specification serves as a reference point.

System Analysis and Evaluation

- Formal methods have been applied to systems that are already built. Some have revealed serious bugs which in turn revealed unstated assumptions, inconsistencies, and the unintentional incompleteness of the system.

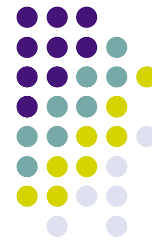
Complexity and Scale

- The problem of scale exists in two dimensions:
  - the size of the specification,
  - the complexity of the specificand.
- Tools can help address specification size.

Complexity and Scale

- A specificand's inherent complexity results from internal complexity and/or interface complexity. By providing a systematic way to think and reason about specificands, formal methods can help people cope with both kinds of complexity.

Formal Methods within the Development Lifecycle



Levels of Formal Method Application

- In a 1993 paper, Rushby defined four levels of rigour in the application of formal methods.

Level 0 - No Use of Formal Methods

- Documents written in natural languages, pseudocode, diagrams and equations.
- Verification is a manual process of review and inspection.
- Validation is based on testing determined by the requirements, specifications and program structure.

Level 1 - Use of Concepts and Notation from Discrete Math

- Some of the natural language components of requirements and specifications are replaced with notations and concepts derived from logic and discrete math.
- Proofs, if any, are performed informally.

Level 1 - Use of Concepts and Notation from Discrete Math

- Advantages
  - compact notation that can reduce ambiguities
  - provides a systematic framework that can aid software development

Level 2 - Use of Formalized Specification Languages with some Automated Support Tools

- Specification languages provide
  - standardized notation for discrete math
  - some automated methods of checking for certain classes of faults
- Proofs are conducted informally (rigorous proofs).
- Formal proofs are possible but manual.

Level 3 - Use of Fully Formal Specification Languages with Comprehensive Support Environments, including Automated Theorem Proving or Proof Checking

- Use of specification languages that employ a strictly defined logic and provide techniques for the use of formal proofs.

Level 3 - Use of Fully Formal Specification Languages with Comprehensive Support Environments, including Automated Theorem Proving or Proof Checking

- Use of proof checkers and theorem provers
  - proof checkers: check the steps of a proof produced by an engineer or designer
  - theorem provers: attempt to discover proofs without human assistance

Level 3 - Use of Fully Formal Specification Languages with Comprehensive Support Environments, including Automated Theorem Proving or Proof Checking

- Advantages
  - greatly increases the probability of detecting faults within the various descriptions of the system
  - automated proof techniques remove the possibility of faulty reasoning

Industrial Applications of Z

- IBM's CICS
  - Development of the CICS transaction processing system by IBM.
  - New software release
    - over a quarter of a million lines of new code
    - 37,000 lines produced from Z specs
    - 11,000 lines partially specified in Z
  - Formal specifications were subjected to rigorous verification.

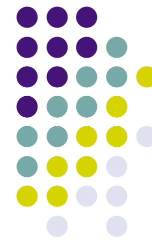
Industrial Applications of Z

- IBM's CICS
  - IBM estimated that formal methods reduced the number of problems per line of code by a factor of 60%.
  - Reduced code production cost by 9%.

Industrial Applications of Z

- INMOS T800
  - Development in Z of the floating-point unit for the T800 Transputer by Inmos and the University of Oxford.
  - Uncovered faults in the IEEE floating-point standard and in other hardware implementations used for testing purposes.
  - Inmos estimated that the development work was completed in less than 50% of the time required for informal methods.

Formal Specifications in Z



Z Schemas

- A specification in Z is a collection of schemas.
- A schema contains specification entities and the relationships between these entities.

Z Schemas

- Parts of a schema include:
  - The top line of the schema contains the schema name.
  - Below the top line and above the dividing line is the signature where the names and types of the entities are introduced.
  - The predicate (bottom part) sets out the relationships between the entities in the signature by defining a predicate over the entities which must always hold.

Z Schemas

- The effect of combining specifications is to make a new specification which inherits the signatures and predicates of the included specifications.
- These inherited signatures and predicates are combined with any new signatures and predicates which are introduced in the new specification.

Z Schemas

- The predicates are written on separate lines and an implicit and separates them.

Container

contents: N

capacity: N

---

contents <= capacity

Z Schemas

Indicator

light: {off,on}

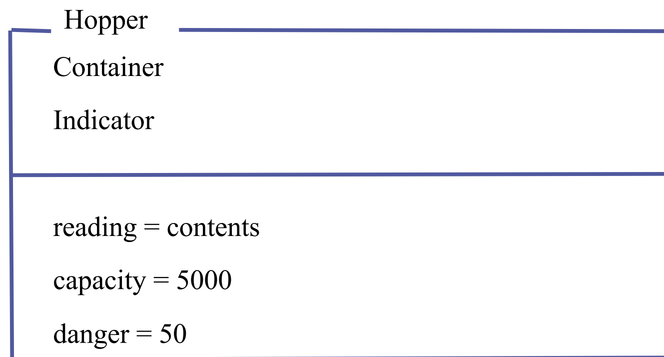
reading: N

danger: N

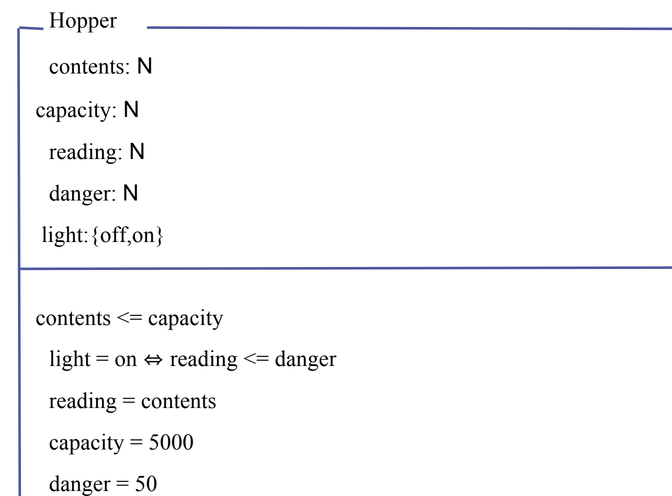
---

light = on ⇔ reading <= danger

Z Schemas



Z Schemas



Operations

```
FillHopper
ΔHopper
amount?: N
--------------------------------
contents' = contents + amount?
```

- Names whose final character is a ? are always taken to indicate operation inputs.
- The values of an entity after an operation are referenced by adding the suffix quote mark (') to the entity name.

Delta Schema

- The delta schema indicates that the effect of the operation is likely to change one or more values of the schema's entities.

```
ΔHopper
Hopper
Hopper'
```

We have all the declarations of Hopper and the predicate.

We also have another set of declarations for a corresponding set of dashed variables and the equivalent predicate.

Xi Schema

ΞHopper

ΔHopper

contents = contents’ ∧ capacity = capacity’ ∧

light = light’ ∧ reading = reading’ ∧

danger = danger’

- Shorthand to say that everything in a schema remains the same.

SafeFillHopper

ΔHopper

amount?: N

contents + amount? <= capacity

contents' = contents + amount?

Output

```
   OverFillHopper
ΔHopper
amount?:  N
r!: seq CHAR
-----------------------------------
capacity < contents + amount?
contents = contents’
r! = "Hopper overflow"
```

- Names whose final character is an ! are always taken to indicate operation outputs.

The OR Operator

```
   FillHopperOp
SafeFillHopper V OverFillHopper
```

The OR Operator

FillHopperOp

ΔHopper

amount?: N

r!: seq CHAR

(contents + amount? <= capacity

contents' = contents + amount?)

V

(capacity < contents + amount?

contents = contents'

r! = "Hopper overflow")

Specification Using Functions

- Functions can be used to model data structures.
- A schema can define a partial function by use of the tagged arrow indicator. Then, given a name, the associated type and description can be discovered.

Specification Using Functions

- The enclosure of the schema name in curly brackets defines a set.
- The name of the set is in upper-case characters.
- The name of a partial function is used in the same way as a function name in a programming language, with the name acting as a parameter.

Specification Using Functions

DataDictionary

DataDictionaryEntry

ddict: Name ⇸ {DataDictionaryEntry}

Functions

- A **function** is an abstraction over an expression in programming languages.
- In *Z*, a function is a set of pairs where each pair shows how an output relates to an input.
- A **partial function** is a function where not all possible inputs have a defined output.

Functions

- The **domain** of a function is the set of inputs over which the function has a defined result.
- The **range** of a function is the set of results which the function can produce.
- If an input i is in the domain of some function f(i $\in$ dom f), the associated result may be specified as f(i) , i.e. f(i) $\in$ rng f.

Functions

DataDictionaryEntry

ident: NAME

type: {process,dataflow,datastore,uinput,uoutput}

description: seq CHAR

#description <= 2000

Functions

GetDescription

DataDictionary

name?: NAME

desc!: seq CHAR

---

name? ∈ dom ddict

desc! = ddict(name?).description

Functions

DeleteEntry

ΔDataDictionary

name?: NAME

---

name? $\in$ dom ddict

ddict' = {name?} $\lhd$ ddict

Functions

MakeNewEntry

ΔDataDictionary

name?: NAME

entry?: DataDictionaryEntry

---

name? $\notin$ dom ddict

ddict' = ddict $\cup$ {name? $\mapsto$ entry?}

Functions

ReplaceEntry

ΔDataDictionary

name?: NAME

entry?: DataDictionaryEntry

---

name? ∈ dom ddict

ddict' = ddict ⊕ {name? ↦ entry?}

ddict(name?).type = entry?.type

Functions

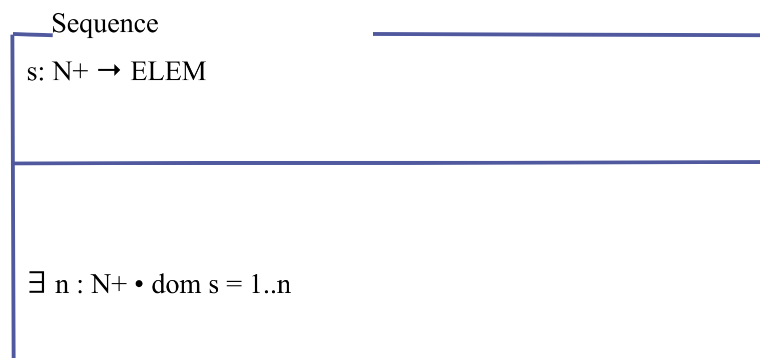
AddDictionaryEntry

MakeNewEntry ∨ ReplaceEntry

Specification Using Sequences

- A sequence is a collection where the elements are referenced by their position in the collection
- Formally, a sequence of X is a mapping where the positive integers have associated values in X and the domain of the mapping includes all integers from 1 to n where n is the length of the sequence.

Specification Using Sequences



Specification Using Sequences

NewDataDictionary

DataDictionaryEntry

ddict: seq {DataDictionaryEntry}

$\forall$ i,j: dom ddict • s(i).ident < s(j).ident