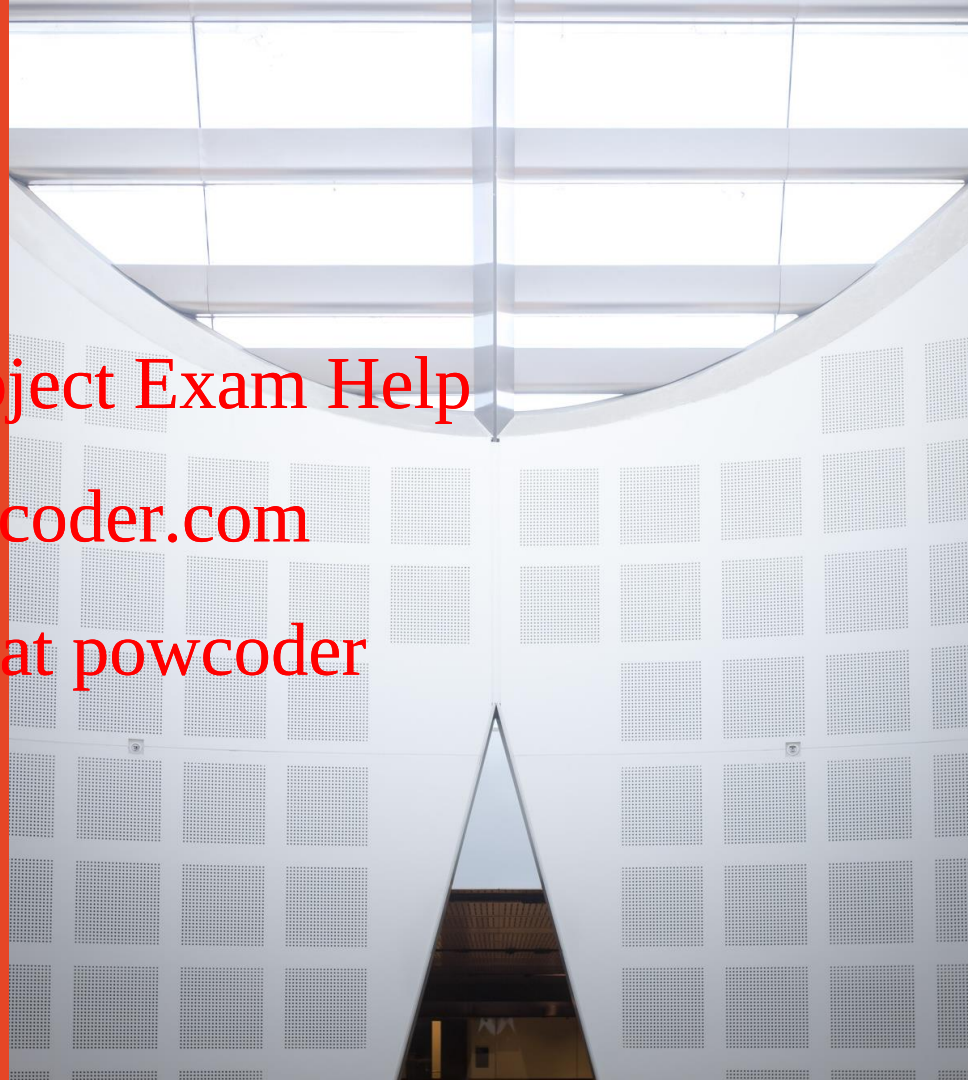
Software Design and Construction 2
SOFT3202 / COMP9202

Software Verification / Specification Languages

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Agenda

- Formal Methods
- Formal Specification Languages
  - Z Specifications
- Specification-based testing
  - Decision Tables
  - State Transition

Formal Methods

Theory

THE UNIVERSITY OF SYDNEY

Formal Methods

- Broadly two domains:
  - Formal specifications (spec.)
    - Write precise unambiguous specifications
    - Code spec. and design spec.
  - Formal verification
    - Prove code and abstract systems are correct
    - Code verification and design verification

- Some confusion in the community, sometimes
  - Formal specification refers to specify and verify systems
  - Formal verification refers to specify and verify code

Verification and Specification – Terminology

- Partial verification
  - Only verify a subset of the specification
  - E.g., "it never crashes or accepts the wrong password"
- Full verification
  - Verify the entire specification
  - E.g., "it never crashes or admits the wrong password and locks the account if you give the wrong password three times."
- Type of software
  - Mission-critical (high-assurance) software
  - Other software

Code Specifications

- Clear code specifications (what the code should do) is required to prove the code is correct
  - Which is not ambiguous?
    - "A list should be sorted" OR
    - "A list of integers n is sorted in ascending order if for any two indices I and j, if I < j, then n[i[<= n[j]

Code Specifications – Classifications

- Statements independent of the code
  - E.g., "this function returns sorted lists"
- Embed specification in the code (Design by Contract)
  - Pre/postconditions, assertions and invariants
  - Most popular of verification
- Type-systems
  - Any math theorem or proof can be encoded as dependent type
  - E.g., define type of "sorted lists" as [Int] -> Sorted [Int]
- Maps to the main domains of automated correctness checking
  - Tests Contracts and Types

What's the Right Spec.

- The goal of code verification is to prove that it meets the spec. but how do we verify the correctness of the spec.?
  - If it does what the user wants (validation)?
    - Does the user know what they really want?
      - Rapid iterations should help to find out
      - We can assume what the users don't want, e.g., software does not crash, does not have security holes
- Requirements are often thought of in human terms not mathematically
- The challenges is how to formalize human concepts
- Having right spec will help to prove the code meets the spec.

Proving the Spec

- Dijkstra-style – "think really hard why it's true"
- E.g., to prove insertion sort works by thinking:
  - *Base Case*: if we have an empty list and add one element to it, that will be the only element, so it will be sorted.
  - If we have a sorted list with k elements and add one element, we insert the element so that it is after all smaller numbers and before all larger numbers. This means the list is still sorted.
  - By induction, insert sort will sort the entire list.
- But, "Beware of bugs in the above code; I have only proved it correct, not tried it." – Knuth quote

Proofs are hard

- For example to prove the Induction in the insertion sort example
  - Need to formalize what induction is, how it works and how it is valid
  - Formalize every assumption
  - Write the prove (or use prover)
    - This needs good background in Math, computer science, domain knowledge, details of program and spec.
- Programming language may make proofs more difficult
  - Assuming addition is associative can be dangerous as some languages such as C++ are not associative
    - INT_MAX((-1) + INT_MAX) + 1 is INT_MAX -1 + (INT_MAX+1) is not defined

Proofs are hard

- For example to prove the Induction in the insertion sort example
  - Need to formalize what induction is, how it works and how it is valid
  - Formalize every assumption
  - Write the prove (or use prover)
    - This needs good background in Math, computer science, domain knowledge, details of program and spec.
- Programming language may make proofs more difficult
  - Assuming addition is associative can be dangerous as some languages such as C++ are not associative
    - INT_MAX((-1) + INT_MAX) + 1 is INT_MAX -1 + (INT_MAX+1) is not defined

Verification – Trade-offs

- Factors to consider how rigorously to verify the code, cost and time of verification
- The more the code is verified the better but that costs more time and money
- Other constraints to optimize include performance, time to market, regulations
- Optimum isn't necessarily "fully proved correct"
- What's the minimal required verification? And how much does it cost?
  - E.g., 95% correct. How much would it cost to make it 98% correct?

Verification – Trade-offs

- Use proper types and testing can improve correctness but sometime doesn't make enough verification
- Developer practices can help to get good verification, e.g., Cleanroom develop practices includes
  - Comprehensive documentation
  - Careful flow analysis
  - Extensive code reviews
  - No proofs, no formal verification

Proofs and Programming Languages

- Most languages have positive features that impede proofs

```
a = true;
b = false;
f(a);
assert a;
```


- Is the above code always true? It depends
  - The function f may modify a
  - Another thread concurrency may modify a
  - b maybe aliased to a
- If the language supports any of the above, then you need to explicitly prove they do not occur (this make proofs harder)

Design Specification/Verification

- Design Verification is more about components and their interactions
- We try to formalize the intentions of the overall system not the code
- Example: code procedure/specification "*if called, it makes a system call to persist data and handle system errors*" properties to verify include:
  - Does it serialize data properly?
  - Do malformed inputs violate our guarantees?
  - Do we handle all possible ways the system call could fail?
- This would be different when compared to high-level system spec. as per the example in next slide

Design Verification

- Example: design spec. “all messages are logged” require verifying:
  - All messages, or all messages that reach the system? Are messages logged once or exactly once?
  - How are messages being sent? Is it a queue? Does the transfer medium deliver once? Does it deliver in order?
  - By “logged”, do we mean “permanently logged?” Is the message allowed to be logged and later unlogged? Is it allowed to “bounce” between logged and unlogged before ending logged?
  - What if the server explodes in the middle of logging the message? Do we need journaling?
  - Are there any properties of the storage medium that matter? Is “the medium loses data” outside the scope of our requirements or not?
  - etc.

Design Specification

- Formal specification allows expressing our intentions about the software design (what we actually need the system to do)
- Formal specification ensures that we are actually building what we need to build

Specification Languages

- Specification (or design) languages are for means for represent designs formally
- Like programming languages for code
- Practitioners claim that spec. languages provide insight into problems and makes it easier to explore solutions
- Also, could help designers to work faster and reduce cost of writing specs as it would help discover design mistakes early

Correct-by-construction https://www.youtube.com/watch?v=03mUs5NlT6U

Specification Languages

- There's huge variety of different spec languages which are influenced by specific problem domains

Language	modelled	Use
Z	Business Requirements	Relational Algebra
Promela	Messaging	CSP
SDL	Telecommunications	Flowcharts
Harel Statecharts	Controllers	Automata
Decision Tables	Decisions	Tables

Formal Specification Languages

Formal Specification Languages

- The aims:
  - Specify requirements formally
  - Analyze the problem formally
  - Implement by correctness-preserving transformations
  - Maintain the specification, no longer the code

- **Formal** means requirements are defined using formal syntax and semantics
- Typical forms include:
  - Purely descriptive (e.g., algebraic specification)
  - Purely constructive (e.g., Petri nets
  - Model-based hybrid forms (e.g., OCL, B, Z)

Adapted from

Formal Specification Languages

- **VDM** – Vienna Development Method (Björner and Jones 1978)
- **Z** (Spivey 1992)
- **OCL** (from 1997; OMG 2012)
- **Alloy** (Jackson 2002)
- **B** (Abrial 2009)

Adapted from

Algebraic Specification

- Originally designed in 1977 for specifying complex data
- The syntax defined by the signature of operations
- The semantic defined by axioms which describe properties that are invariant under execution of operations (i.e., expressions being always true)
- Purely descriptive and mathematically well-designed
- Not easy to read and understand
- Rarely (if not) adopted in practice/industry

Adapted from

Algebraic Specification – Example (1)

- Specifying a stack (LIFO) data structure
- Let *bool* be a data type with a range of {false, true} and *boolean* algebra as operations. Further, let elem be the data type of the elements to be stored.

TYPE Stack

FUNCTIONS

new:	()	→	Stack;	-- Create new (empty) stack
push:	(Stack, elem)	→	Stack;	-- add an element
pop:	Stack	→	Stack;	-- remove most recent element from stack
top:	Stack	→	elem;	-- returns most recent element
empty:	Stack	→	bool;	-- true if stack is empty
full:	Stack	→	bool;	-- true if stack is full

Algebraic Specification – Example (2)

AXIOMS

∀ s ∈ Stack, e ∈ elem

(1) ¬ full(s) → pop(push(s,e)) = s -- *pop* reverses the effect of push

(2) ¬ full(s) → top(push(s,e)) = e -- *top* retrieves the most recently stored element

(3) empty(new) = true -- a *new* stack is always empty

(4) ¬ full(s) → empty(push(s,e)) = false -- after *push*, a stack is not empty

(5) full(new) = false -- a *new* stack is not full

(6) ¬ emtpy(s) → full(pop(s)) = false -- after *pop*, a stack is not full

Model-based Formal Specification

- Mathematical model of system state and state change
- Based on sets, relations and logic expressions
- Typical language elements
  - Base sets
  - Relationships (relations, functions)
  - Invariants (predicates)
  - State changes (by relations or functions)
  - Assertions for states

Adapted from

Z Specification

- Model-based formal language for describing computer programs and computer-based systems
- Originally proposed in 1977 by Abrial with the help of Steve Schuman and Bertrand Meyer
- Developed further at the Programming Research Group a Oxford University
- Z named as “it is the ultimate language!”. It is also associated with Zermelo due to the use of Zermelo-Fraenkel set theory
- Used in transaction processing project at IBM Hursely

Adapted from

Z Specification

- Based on mathematical notations including set theory, lambda calculus and first-order predicate logic
- All expressions are typed (to avoid inconsistencies of naïve set theory)
- Commonly used mathematical functions and predicates are defined using Z and available as *mathematical toolkit*

Z Specifications – Basic Elements (1)

- Specification consists of sets, types, axioms and schemata
- Types are elementary sets:
  - *E.g.*, **IN** is set of natural numbers
- Sets have a type:
  - Counter: **IN**
- Axioms define global variables and their (invariant) properties

string: ***seq*** *CHAR* Declaration

#string ≤ *64* Invariant

seq: sequence of elements

#string: number of elements of set string

Z Specifications – Basic Elements (2)

- Schemata
  - organize a Z-specification
  - constitute a name space

Name

Counter
Value, Limit: IN
Value ≤ *Limit* ≤ 65535

Declaration part:

Declaration of state variables

Predicate part:

- Restrictions
- Invariants
- Relationships
- State change

Z-Specifications – Relations

- Relations and functions are ordered set of tuples:

Order: $\mathbb{P}$ (*Part* x *Supplier* x *Date*)

A subset of all ordered triples (p, s, d) with p ∈ *Part*, s ∈ *supplier,* and d ∈ *Date*

$\mathbb{P}$ Power set (set of all subsets) of *M*

Z-Specifications – Relations

- Relations and functions are ordered set of tuples:

$Order: \mathbb{P}\ (Part \times Supplier \times Date)$

$Birthday: Person \rightarrow Date$

A subset of all ordered triples (p, s, d) with p ∈ *Part*, s ∈ *supplier,* and d ∈ *Date*

A function assigning a date to a person, representing the person's birthday

$\mathbb{P}$ Power set (set of all subsets) of *M*

Z-Specifications – State Changes

- State change through operations:

Increment counter

Δ *Counter*

Value < *Limit* *Value'*
= *Value* + *1* *Limit'*
= *Limit*

ΔS The sets defined in schema S will be changed

M' State of set M after executing the operation

Mathematical equality, no assignment!

Z Specifications – Library System

- The library has a stock of books and a set of persons who are library users.
- Books in stock may be borrowed.

Library

$Stock: \mathbb{P}\, Book$

$User: \mathbb{P}\, Person$

$lent: Book \pfun Person$

$\mathbf{dom}\ lent \subseteq Stock$

$\mathbf{ran}\ lent \subseteq User$

$\pfun$	Partial function
dom	Domain ...
ran	Range... ...of a relation

Z Specifications – Operators

- Logical operators
  - negation ¬
  - Conjunction ∧
  - Disjunction ∨
  - implication ⇒
  - equivalence ⇔
- Equality
  - equality = (on all types but not predicates)

Z Specifications – More Notations

- Sets, Sets operations
- Types: pre-defined, free dictionary types, compound
- Variables
- Axiomatic definitions
- Relations
- Functions
- Finite constructs (finite sets)
- Schemata

Useful summary slides of Z specifications and notations https://formal.iti.kit.edu/~beckert/teaching/Spezifikation-SS04/11Z.pdf

Specification-based Testing

THE UNIVERSITY OF SYDNEY

Specification-based Techniques

- Black-box testing (verification) based on specifications
  - Equivalence partitioning
  - Boundary analysis
  - Decision tables
  - State transition

Decision Tables

- Technique for identifying test cases based on combination of things (e.g., inputs)
- Known as cause-effect table
- Easy to understand representation
- Can support automated/manual test case generation
- Useful for certain systems, e.g., control systems

Decision Tables

- Function/subsystem behave based on combination of inputs/events
  - Subsets if it's too large
- Construct the decision table
  - Conditions
  - Rules all combinations of T and F for each aspect
  - Actions/results

Decision Tables – Structure

Conditions	R1	R2	R3	Rm
C1				
C2				
Cn				
Actions/Outcomes				
A1				
A2				
Ai				

Decision Tables – Example

Conditions	R1	R2	R3	R4
Payment has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F

Decision Tables – Example

Conditions	R1	R2	R3	R4
Payment has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	

Decision Tables – Example

Conditions	R1	R2	R3	R4
Payment has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount?	Y	Y		
Process term?	Y		Y	
Error?				Y

Decision Tables – Example

Conditions	R1	R2	R3	R4
Payment has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount?		Y		
Process term?			Y	
Error?	Y			Y

Decision Tables – Example

Conditions	R1	R2	R3	R4
Payment has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Results	Error message	Process loan amount	Process term	Error message

State Transition Verification

- Verification technique in which aspects of the system is represented as '*finite state machine*'
- System in finite number of different states and transitions determined by the machine rules
- Often modelled as state diagram
- The model can be as detailed or as abstract as needed
  - Important part of the system requires more testing and hence modelled in detail

State Transition – Basic Structure

- States of the system (e.g., open/closed connection)
- Transitions from a state to another (not all transitions are permitted)
- Events that trigger a transition (withdraw money change the account state)
- Actions that results from a transition (error message or desired results)

State Transition – State Changes

- Withdrawing $300 from an ATM result in giving cash if sufficient cash available
- Withdrawing the same amount again may result in error message due to insufficient funds
- State: sufficient funds → insufficient funds

State Transition – Example

- Withdrawing $300 from an ATM result in giving cash if sufficient cash available
- Withdrawing the same amount again may result in error message due to insufficient funds
- State: sufficient funds → insufficient funds

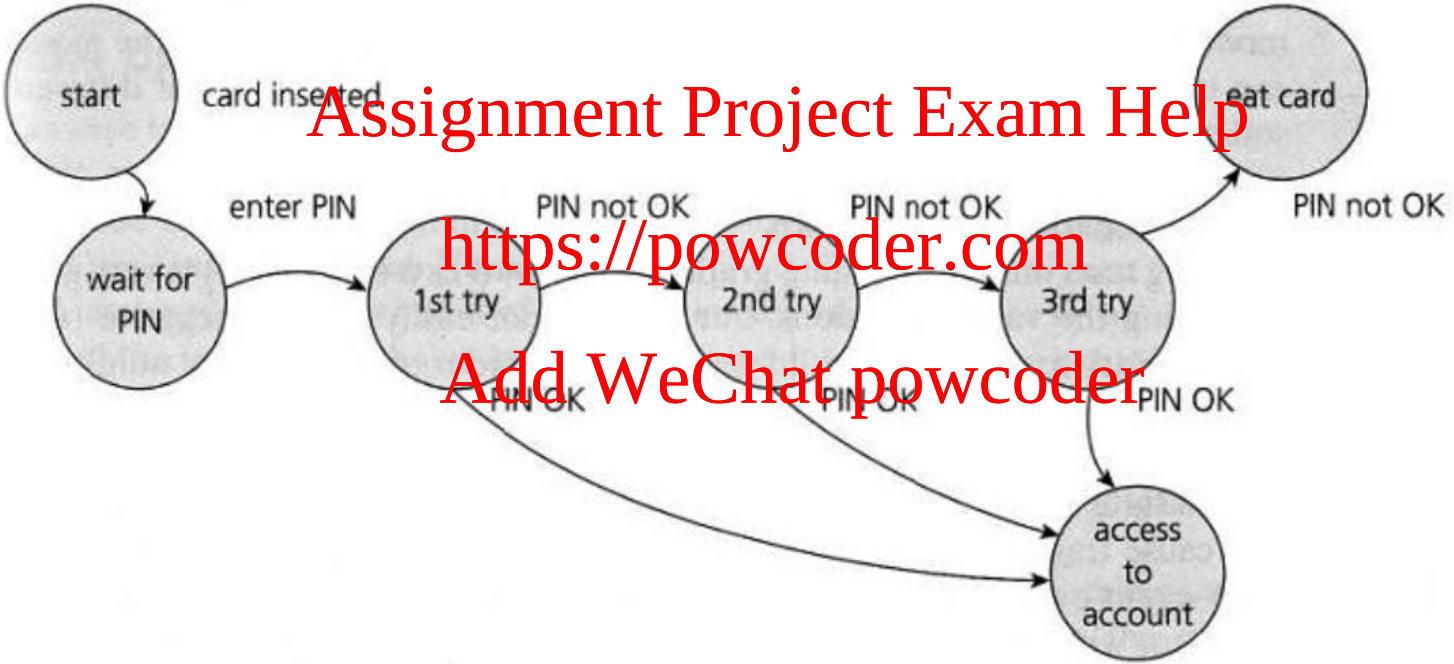
State Transition – PIN Bank Account

- Enter and verify PIN to a bank account
  - The customer inserts a valid bank card
  - The customer is prompted to enter their PIN
  - The customer can enter their PIN up to 3 times
  - After three incorrect PIN trials, the card will be locked by the ATM
  - Entering correct PIN, with trail limit, results in accessing the bank account

PIN Bank Account – Analysis

- Enter and verify PIN to a bank account
  - States
    - Start, wait for PIN, 1st try, 2nd try, 3rd try, eat card, access to account
  - Events
    - Card inserted, Enter PIN, PIN OK and PIN not OK
    - Other?
  - Transitions
    - First try state to second try state in case of invalid PIN
  - Actions
    - e.g., message “please enter your PIN”

PIN Bank Account – State Diagram



PIN Bank Account – Deriving Test Cases

- Possible test cases include:
  - TC1: Correct PIN entered first time
  - TC2: Enter Incorrect PIN each time and the system eats the card
  - TC3: PIN incorrect first time, correct second time
  - TC4: PIN incorrect in the first and second, but correct in the third time

- Test conditions from the state diagram
  - E.g., Each state/transition can be a test condition

State Transition – Coverage Criteria

- Possible coverage measures:
  - State coverage (% of visited states)
  - Valid transitions exercised
  - Pairs of valid transitions exercised
  - Invalid transitions exercised

Unit of Study Surveys

THE UNIVERSITY OF SYDNEY

Unit of Study Feedback 2019

- To share what you enjoyed and found most useful in your learning, and to provide constructive feedback
- To 'pay it forward' for the students coming behind you, so that their **learning experience** in this class is as good, or even better, than your own.
- When you complete your USS survey, please:
  - **Be relevant**
  - **Be specific**
    - Which class tasks, assessments or other activities helped you to learn? *Why* were they helpful?
    - Which one(s) *didn't* help you to learn? *Why* didn't they work for you?
  - **Be constructive**
    - What practical changes can you suggest to class tasks, assessments or other activities, to help the next class learn better?

Unit of Study Survey 2019

- Complete the ONLINE survey at
  - https://student-surveys.sydney.edu.au/students/complete/form.cfm?key=uss203184

- each survey completed will give you an entry into a prize draw to win a range of Apple products including:
  - 64gb Apple iPad Pro 10.5-inch
  - 128GB Apple iPad mini
  - 4 and JB HiFi Gift Cards

Exam Information and Preparation

To be presented in w13 lecture

THE UNIVERSITY OF SYDNEY

References

- Hillel Wayne, Stamping on event streaming, https://www.hillelwayne.com/post/stamping-on-eventstream/
- Z Notations, Wikipedia, https://en.wikipedia.org/wiki/Z_notation
- Lionel Briand, Blackbox, Functional Testing, https://www.uio.no/studier/emner/matnat/ifi/nedlagte-emner/INF4290/v11/undervisningsmateriale/INF4290-BBT.pdf
- TryQA, what is decision table in software testing http://tryqa.com/what-is-decision-table-in-software-testing/
- TryQA, what is state transition in software testing http://tryqa.com/what-is-state-transition-testing-in-software-testing/