Introduction to Formal Methods in Testing

Getinet Yilma, Ravindra Babu

Overview

Specification language in Formal Methods

- Non-formal
  - Natural language, Flow chart, Decision tree, Pseudo code
- Semi-formal
  - UML
- Formal methods
  - Precisely defined semantics
  - Algebraic expression, Alloy, Z, B, and VDM
  - mechanisms for abstraction, analysis…

Formal specifications

- Specifications given in some formal, mathematical notation.
- The syntax and semantics of that notation is precisely and unambiguously defined.
- If a Software functionality is quite complex FM is advisable.
- Formal specification languages thus allow the precise, unambiguous expression of requirements using formal semantics.
- Most formal specification languages offer mechanisms like abstraction, modularity and reuse. These mechanisms aid the specifier in controlling large and complex requirements.
- The process of expressing requirements formally helps uncovering ambiguities, inconsistencies, or incompleteness.

Writing formal specifications

Natural language

Ambiguities

- For up to 12 aircraft, the small display format shall be used. Otherwise, the large display format shall be used

"up to 12". It is not clear whether it includes 12 or not.

- "Aircraft that are non-friendly and have an unknown mission or the potential to enter restricted air-space within 5 minutes shall raise an alert."

The problem here is that the relative precedence of "and" and "or" is unclear.

- "All members have a unique membership number".

The problem is that we do not know whether every member has its own unique number, or whether all members share the same unique number.

Writing formal specifications

- **Natural language**

Ambiguities

"Everybody likes a holiday." it is not clear whether everybody likes the same holiday (∃ h. ∀ x .likes(x , h)),
or
if people may have different holidays that they like (∀ x . ∃ h.likes(x , h)).

- The problem is that natural language is full of semantic ambiguities
- Natural language cannot be analyzed automatically instead may take contexts
- Prepositional and Predicate logic is very expressive, well-studied and understood

Writing formal specifications

- Operation add on dictionaries modeled as elements of P(Key × Value), i.e., sets of Key, Value pairs: **add : Key × Value × P(Key × Value) → P(Key × Value)**
- The effect of executing add on an existing dictionary d using the key k and the value v can be captured by the following specification:

$$\forall\, d : \mathcal{P}(Key \times Value).\, \forall\, d' : \mathcal{P}(Key \times Value).\, \forall\, key : Key.\, \forall\, val : Value.$$
$$d' = add(k, v, d) \quad \leftrightarrow$$
$$\big((\neg\ \exists\, v' : Value.\langle k, v'\rangle \in d) \quad \rightarrow \quad d' = d \cup \{\langle k, v\rangle\}\big) \wedge$$
$$\big(\exists\, v' : Value.\langle k, v'\rangle \in d. \quad \rightarrow \quad d' = d - \langle k, v\rangle \cup \langle key, val\rangle\big)$$

- The specification says that a dictionary d' is the result of adding a key k and a value v to some other dictionary d if and only if d' is equal to d ∪ {(key, v)} if k is not already used, and otherwise d' is just like d except that the old pair with k is removed and the new pair {k, v} added.
    - Predicate logic has less support for modularity
    - large specifications in predicate logic usually are very hard to understand

Writing formal specifications

- More readable
- Modular

operation $add(k,v,d)$

input

$k : Key$

$v : Value$

$d : Dictionary\ over\ Key \times Value$

output

$d' : Dictionary\ over\ Key \times Value$

---

pre-condition

$true$ $\%$ *no constraints on the input*

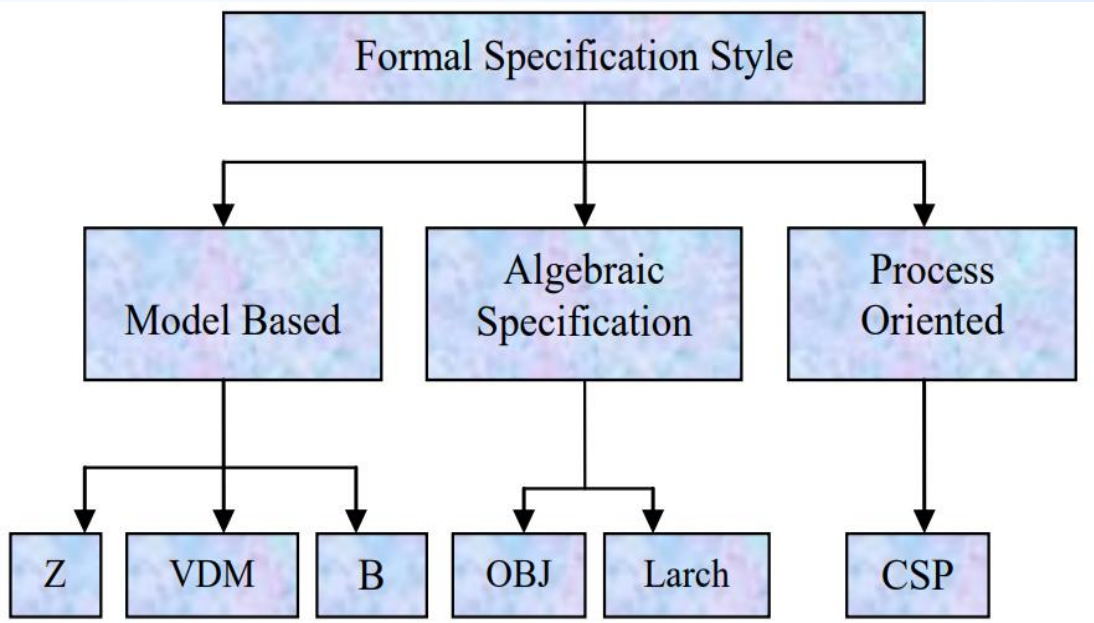
post-condition

$\Big((\neg\ \exists\, v' : Value\ .\ \langle k, v' \rangle \in d) \quad \rightarrow \quad d' = d \cup \{\langle k, v \rangle\}\Big) \wedge$ $\%$ *case 1*

$\Big(\exists\, v' : Value\ .\ \langle k, v' \rangle \in d. \quad \rightarrow \quad d' = d - \{\langle k, v \rangle\} \cup \{\langle key, val \rangle\}\Big)$ $\%$ *case 2*

Formal Modeling Methods

- Formal methods are mathematics based languages, techniques and tools that can be applied at any part of the program lifecycle.
- Formal methods make use of refinement techniques at any stage to ensure the correctness, completeness and consistency of specification.
- The representation used in formal methods is called a formal specification language
- A formal specification language can be used to model the most complex systems using relatively simple mathematical entities, such as sets, relations and functions.
- Types of Formal Specification Styles



Formal Modeling Methods

1. **Model Based Languages** : Specify system behavior by the construction of a mathematical model with an underlying state (data) and a collection of operations on that state using relations, sets, sequences and functions.
   - The most widely used notations for developing model based languages are **Vienna Development Method (VDM)** , **Zed (Z)** and **B.**
2. **Algebraic Specification:** These languages uses methods derived from abstract algebra to specify behavior of information system.
   - Algebraic approach was originally designed for the definition of abstract data types and interface.
   - The most widely used notations for developing algebraic specification languages are **LARCH**, **ASL** and **OBJ**.
3. **Process Oriented Specification** is used to describe concurrent system.
   - The most widely used process oriented language is Communicating Sequential Processes (CSP).

Model based specification

- **Model-based specification** is an approach to formal specification where the system specification is expressed as a system state model.
- The state model is constructed using well-understood mathematical entities such as sets and functions.
- System operations are specified by defining how they affect the state of the system model.
- The most widely used notations for developing model-based specifications are Z, CSP, B, Petri Net and VDM.

	Sequential	**Concurrent**
Algebraic	OBJ(1985), CASL (Common Algebraic Specification Language),	CCS (Calculus of Communicating Systems), ACP (Algebra of Communicating Processes)
Model	Z, VDM, B	CSP (Communicating Sequential Processes), TLA+ (Temporal Logic of Actions)

Model- Based Languages

Z-method

- Z formalism goal was to develop a specification formalism that achieves precision, conciseness, and modularity without imposing unnecessary constraint.
- Z is based on first-order predicate logic and set theory. Its built-in theory contains booleans, integers, reals, relations, functions, sets, bags (sets with duplicates), and sequences and the standard operations on them.
- Schemas encapsulate specifications of initial states, constants, types, functions, and operations.
- Schemas can be composed using Z's schema calculus.
- A refinement calculus allows the stepwise refinement of specifications into code.
- Z decomposes specifications into manageably sized modules, called schemas:

Model- Based Languages

Z-method…

- Schemas are divided into 3 parts:
  - A State
  - A collection of state variables and their values
  - There are also some operations that can change its state

Model- Based Languages

Z-method…

Name

Some variables are declared. As well as state transitions.

Relationship between the values of the variables

[NAME; DATE]:

Init Birthday Book

Birthday Book

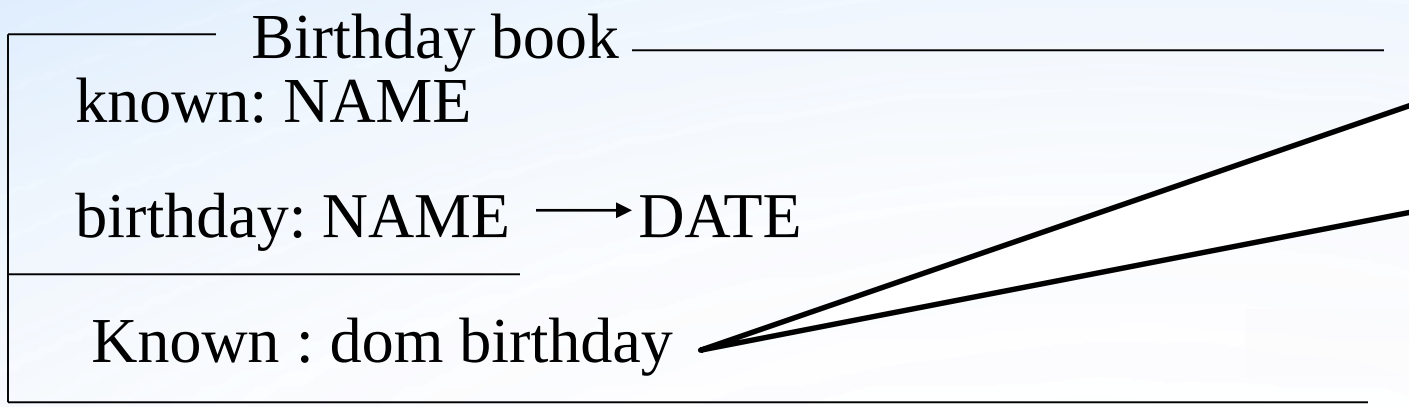
Known = ∅

- **known** is the set of names with birthdays recorded;
- **birthday** is a function which, when applied to certain names, gives the birth-days associated with them.

- This schema describes a birthday book in which the set **known** is **empty**: in consequence, the function **birthday** is **empty** too

Model- Based Languages

Z-method…



It says that the set **known** is the same as the domain of the function birthday { the
set of names to which it can be validly applied.

- One possible state of the system has three people in the set **known**, with their birthdays recorded by the function **birthday**:

known = { John; Mike; Susan }
birthday = { John 25-Mar,
Mike 20-Dec,
Susan 20-Dec }

Model- Based Languages

Z-method…

Add Birthday

△ Birthday Book

name?: NAME

date?: DATE

---

name? ∉ known

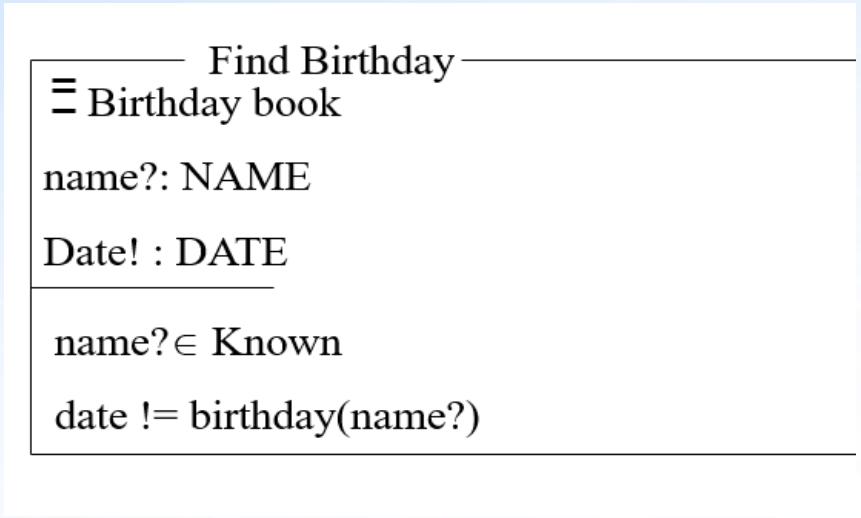
birthday' = birthday U { name? → date}

- The declaration **BirthdayBook** alerts us to the fact that the schema is describing a state change:
- 4 variables The first two are observations of the state before the change, and the last two are observations of the state after the change.
- Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation.
- Next come the declarations of the two inputs to the operation.

- By convention, the names of inputs end in a question mark.
- The part of the schema the lines gives a pre-condition for the success of the operation: the name to be added must not already be one of those known to the system.
- If the pre-condition is satisfied, however, the second line says that the birthday function is extended to map the new name to the given date.

Model- Based Languages

Z-method…



- Another operation might be to find the birthday of a person known to the system. Again we describe the operation with a schema
- The declaration ¥BirthdayBook indicates that this is an operation in which the state does not change where all values are derived from Birthday book

REFERENCES
http://staff.washington.edu/jon/z-lectures/z-lectures.html

Model- Based Languages

Z-method…

Init_Library

Users
Database_book

Registrar≠∅
Reader ≠∅
stock=maxbooks
available=maxbooks
checked_out=∅
book info=∅

Library management system

- Z schema for the Initialization of Library management system The administrator adds a new copy of the book, including the new copy attribute, including the title, author, and subject

AddBook

Δ LIBRARY
title? : Title
authors ? : Author
subject ? : Subject

copy? ∉ stock
stock' = stock ∪ copy?
available' = available ∪ |copy?|
book info' = book info ⊕ |copy? ↦ B|
Where B : Book B.title = title?
B.authors = authors?
B.subjects = subjects?

Z schema for the AddBook

Model- Based Languages

Z-method…

BankAccount

$dollars : \mathbb{N}$
$cents : \mathbb{N}$

$dollars \geq 0$
$cents \geq 0$

Z schema for the Bank System

WithdrawMoney

$\Delta BankAccount$
$dollarAmount? : \mathbb{N}$
$centAmount? : \mathbb{N}$

Bank schema with delta shows there is change

$dollarAmount? \leq dollars$
$dollarAmount? = dollars \Rightarrow centAmount? \leq cents$
$centAmount? > cents$
$\Rightarrow (\quad dollars' = dollars - dollarAmount? - 1$
$\quad \wedge \quad cents' = cents - centAmount? + 100)$
$centAmount? \leq cents$
$\Rightarrow (\quad dollars' = dollars - dollarAmount?$
$\quad \wedge \quad cents' = cents - centAmount? \quad)$

Z schema for the withdraw money

- The system properties above describes the changes made to the amount of dollars and cents

Model- Based Languages

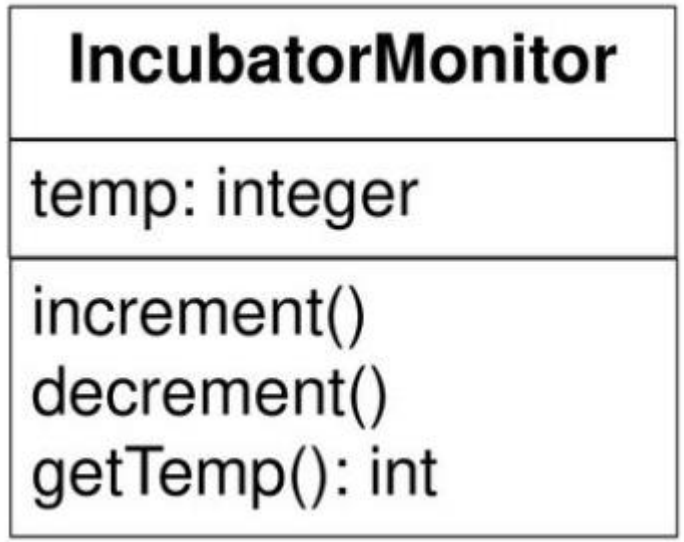
VDM : It is a collection of techniques for the modeling, specification and design of computer-based system.

- A VDM model has a specific role in application areas, such as semantics of programming language, databases and construction of compilers.
- When using VDM, an abstract model traditionally contains the following components.
  - **Semantic domains**: These types describe the objects to be operated on
  - **Invariants**: Invariants are the boolean functions that define a set of condition on the objects that is described by semantic domains.
  - **Syntactic domains**: Types that define a "language" in which to express commands for manipulating the objects defined by the semantic domains.
  - **Well-formedness conditions**: These are the functions that define when the commands, which is defined by the syntactic domains have a well-defined effect.
  - **Semantic functions**: These functions provide the effect of commands on the objects defined by the semantic domains.

Model- Based Languages

Example - Incubator

- The temperature needs to be carefully controlled and monitored in order to provide the correct conditions for a particular biological experiment to be undertaken.
- The temperature of the incubator increments or decrements in response to instructions and each time a change of one degree has been achieved, the software is informed of the change, which it duly records.
- Safety requirements dictate that the temperature of the incubator must never be allowed to rise above 10° C, nor fall below -10 ° C.



State Definition

state ***IncubatorMonitor of***

temp : Z

end

increment()

ext wr ***temp: Z***

pre ***temp < 10***

post ***temp*** $= \overline{temp} + 1$

deccrement()

ext wr ***temp: Z***

pre ***temp > -10***

post ***temp*** $= \overline{temp} - 1$

getTemp() currentTemp: Z

ext rd ***temp: Z***

pre ***true***

post ***temp = currentTemp***

Model- Based Languages

B-method

- The main concept follows in B- method is to initiate with an abstract model of the system under development, which roughly corresponds to the modules in many programming languages.
- A development process creates a number of proof obligations, which guarantee the correctness.
- An abstract MACHINE of a given name contains information given as follows.

MACHINE m
CONSTANTS k
PROPERTIES T
VARIABLES v
INVARIANT I
INITIALIZATION L
OPERATIONS

$$y \leftarrow op(x) \triangleq$$

PRE P THEN S
END;
...
END

Algebraic Specification

1. OBJ

- The OBJ languages are based on algebra, first-order logic, temporal logic and set theory etc…
- Formal manipulation of the specification based classes, objects, inheritance and etc…

2. LARCH: Used to model language dependent and language independent specifications.

Name	Formal Method
TROLL	Logical framework called Object Specification Logic
CSML	Algebraic semantics. Development method based on Jackson System Development
EAM	Entity-Relationships model and Abstract Machines
Disco	Joint Actions and Statecharts
MONDEL	Operational semantics. Formal verification method based on Coloured Petri Nets
OOST	Set theory and Relational Neutral Systems (‘Rest stays unchanged’ semantics)
OOZE	Algebraic semantics derived from the underlying OBJ system
COLD	Operational semantics
Z++	Z-based semantics for objects and operations. Algebraic for classes and inheritance
MooZ	Z (transformational semantics)
LOOPN	Based on Coloured Petri Nets
CO-OPN	Timed Petri Nets and Distributed Transition Systems
VDM++	VDM (transformational semantics)
SDL92	SDL (transformational semantics)

Process Oriented

Communicating Sequential Process(CSP)

- Usually used to model **deadlock**, **live lock**, and **determinism**.
- Csp is used for the formal analysis of **safety-critical systems**, where **deadlock** and **live lock** could have serious consequences.
- It is formal language for describing patterns of interaction in concurrent systems.
- CSP, each process was assigned an explicit name, and the source or destination of a message was defined by specifying the name of the intended sending or receiving process.
- Syntax.
  - CSP has two basic processes: SKIP and STOP. SKIP is used to represent successful termination while STOP represents a deadlocked process which can no longer engage in any events.
  - The syntax of CSP defines the “legal” ways in which processes and events may be combined. Let e be an event, and X be a set of events, then the basic syntax of CSP can be defined as

Process Oriented

Semantics

- formally describe how the individual steps of a computation take place in a computer-based system. By opposition natural semantics (orbig-step semantics) describe how the overall results of the executions are obtained.

$$
\begin{array}{lll}
Proc & ::= & STOP \\
 & | & SKIP \\
 & | & e \rightarrow Proc \quad (prefixing) \\
 & | & Proc \,\Box\, Proc \quad (external\ choice) \\
 & | & Proc \sqcap Proc \quad (nondeterministic\ choice) \\
 & | & Proc \,|||\, Proc \quad (interleaving) \\
 & | & Proc \,|[\{X\}]|\, Proc \quad (interface\ parallel) \\
 & | & Proc \setminus X \quad (hiding) \\
 & | & Proc; Proc \quad (sequential\ composition) \\
 & | & \text{if } b \text{ then } Proc \text{ else } Proc \quad (boolean\ conditional) \\
 & | & Proc \triangleright Proc \quad (timeout) \\
 & | & Proc \,\triangle\, Proc \quad (interrupt)
\end{array}
$$

Formal Verification ATM CSP Example

- Modelling an automated teller machine introduces Csp's *syntax* in an intuitive way and provides a basic understanding of Csp's constructs
- The description of the ATM in natural language

Initially the ATM shows a ready screen. A customer then proceeds to

- *insert a bank card, and*
- *enter the pin for the card*

following which the ATM will

- *deliver cash to the customer, and*
- *return the bank card,*

before getting ready for a new session.

- The example explicitly mentions four events.
  - *cardI* Card Insert
  - *pinE* PIN insert
  - *cashO* Cash draw
  - *cardO* returns the card

Specification of Order of events

$$\Sigma = \{cardI, pinE, cashO, cardO, ready\}$$

Formal Verification ATM CSP Example

- A single session of our simple ATM can be modelled as follows

ATM0 = ready → cardI → pinE → cardO → cashO → Stop

- *ATM0* starts with a *ready* event followed by the events *cardI*, *pinE*, *cardO* and *cashO*.
- Recursion in ATM1 to overcome the single session in ATM0

ATM1 = ready → cardI → pinE → cardO → cashO → ATM1

- ATM1 performs the same activities as ATM0 however after cashO it starts over again

ATM2 = Disply.ready → CardSlot.cardI → KeyPad.pinE →CardSlot.cardO → CashSlot.cashO → ATM2

ATM interfaces ***Display, CardSlot, KeyPad, Cash slot are taken as communication Channels.***

ATM2 displays *ready* message at the *Display*, receives a *cardI* over the CardSlot, etc.

Formal Verification ATM CSP Example

- channel *c* has a set of 'basic' events as its type *T*(*c*), the channel *CardSlot* has the type *T*(*CardSlot*) = *{cardI, cardO}.*
- In Csp it is standard to write

$$events(c) \triangleq \{c.x \mid x \in T(c)\}$$

for a set of events associated with channel c. Given a list of channels $c_1, \ldots, c_n$, $n \geq 1$, their combined events are given by

$$\{|\, c_1, \ldots, c_n \,|\} \triangleq events(c_1) \cup \ldots, \cup events(c_n).$$

Using this notation, we can write the alphabet of $ATM2$ in a structured way:

$$\Sigma = \{|\, Display, CardSlot, KeyPad, CashSlot \,|\}$$

Communicating an event e over a channel c adds to the Csp grammar the following primitives

$$\begin{aligned} P ::= \quad & \ldots \\ & \mid c.e \rightarrow P \end{aligned}$$

Formal Verification ATM CSP Example

Process Termination

- Enrich the ATM example by considering a '**cancel**' button for **Canceling a Session by Interrupt**.
- *Any time after inserting the bank card and before retrieving the cash, the costumer has a choice to cancel the session. Upon cancellation, the ATM returns the bank card and is ready for a fresh session.*

$$
\begin{aligned}
Session &= (KeyPad.pinE \rightarrow Skip) \triangle SessionCancel \\
SessionEnd &= CardSlot.cardO \rightarrow CashSlot.cashO \rightarrow ATM3 \\
ATM3 &= Display.ready \rightarrow CardSlot.cardI \rightarrow Session_9^9 \\
&\quad SessionEnd
\end{aligned}
$$

Formal Verification ATM CSP Example

Initially the ATM shows a ready screen. A customer then proceeds to insert a bank card and enters the pin for the card following which the ATM will offer the choice between cash withdrawal and checking the balance.

In case of cash withdrawal, the ATM will deliver the cash to the customer, return the bank card, and get ready for a new session.

In case of checking the balance, the ATM will display the account balance, return the bank card, and get ready for a new session.

ATM with Customer Choice

$$
\begin{aligned}
ATM5 \quad &= Display!ready \rightarrow CardSlot.cardI \\
&\rightarrow KeyPad.pinE \rightarrow Display!menu \\
&\rightarrow Buttons?x \\
&\rightarrow \textbf{if } x = checkBalance \textbf{ then } Balance \textbf{ else } Withdrawal
\end{aligned}
$$

$$
Balance \quad = Display!accountBalance \rightarrow CardSlot!cardO \rightarrow ATM5
$$

$$
Withdrawal = CardSlot!cardO \rightarrow CashSlot!cashO \rightarrow ATM5
$$

Further ATM

- **UserDialog and PIN Verification in Csp**

$$\begin{aligned} UserDialog = {} & Display.ready \rightarrow CardSlot.cardI \\ & \rightarrow KeyPad.pinE \rightarrow requestPCheck \\ & \rightarrow ((Check.pinOK \rightarrow Services) \\ & \quad \Box \\ & \quad (Check.pinWrong \rightarrow Display.messagePinWrong \\ & \quad \rightarrow UserDialog)) \end{aligned}$$

- **Add Failed PIN Verification after three attempts**

- **Services**

$$\begin{aligned} Services = {} & Display.menu \\ & \rightarrow (CashWithdrawal \Box BalanceCheck) \end{aligned}$$

$$\begin{aligned} BalanceCheck = {} & Buttons.checkBalance \rightarrow Display.accountBalance \\ & \rightarrow CardSlot.cardO \rightarrow UserDialog \end{aligned}$$

$$\begin{aligned} CashWithdrawal = {} & Buttons.withdrawCash \rightarrow CardSlot.cardO \\ & \rightarrow CashSlot.cashO \rightarrow UserDialog \end{aligned}$$

Algebraic specification

Specifications in Software

- Specifications can be formal or informal
  Formal specs are usually expressed mathematically
  Informal specs are usually expressed in natural language
- Most specification languages include explicit logical expressions, so it is very easy to apply logic coverage criteria
- Programmers often include preconditions for their methods
- The preconditions are often expressed in comments in method headers

Example – Saving addresses

// name must not be empty
// state must be valid
// zip must be 5 numeric digits
// street must not be empty
// city must not be empty

Re writing to logical expression

$name != \text{“”} \wedge state \in stateList \wedge zip >= 00000 \wedge zip <= 99999 \wedge street != \text{“”} \wedge city != \text{“”}$

Algebraic specification forms

Algebraic specifications of a system may be developed in a systematic way

- Specification structuring;
- Specification naming;
- Operation selection;
- Informal operation specification;
- Syntax definition;
- Axiom definition.

< SPECIFICATION NAME > (Generic Parameter)

sort < name >
imports < LIST OF SPECIFICATION NAMES >

Informal description of the sort and its operations

Operation signatures setting out the names and the types of the parameters to the operations defined over the sort

Axioms defining the operations over the sort

Introduction

- Defines the sort (the type name) and declares other specifications that are used.

Description

- Informally describes the operations on the type.

Signature

- Defines the syntax of the operations in the interface and their parameters.

Axioms

- Defines the operation semantics by defining axioms which characterise behaviour.

Algebraic specification forms

- **Stacks:** is a "container"

Description
Stack <Elem>: --the specification is parameterized by the data type
--of the elements to be stored in the stack

Signature:

isEmpty : Stack<Elem> → bool --answers "Is the stack empty?"
topOf : Stack<Elem> → Elem --yields item at top of the stack
empty : → Stack<Elem> --yields the empty stack
pop : Stack<Elem> → Stack<Elem> --yields stack obtained by removing top item
push : Stack<Elem> × Elem → Stack<Elem> --yields stack obtained by placing item at top

Axioms:
(1) isEmpty(empty) = true // he stack described by the expression empty has the property of being empty
(2) isEmpty(push(s,x)) = false any stack obtained by pushing some element x onto some stack s is not empty.
(3) topOf(push(s,x)) = x the element x is on top of the stack obtained by pushing x onto any stack s
(4) pop(push(s,x)) = s if we pop a stack obtained by pushing some element x onto some stack s, we get the stack s as the result

Strengths and Weaknesses

The major strengths of Z method are as follows:

- Z provides a strong base for system designing and testing due to its high level support on abstraction.
- Free tools support is available in the market.
- By using Z method, faults can be found early in the development.
- Z is used to specify the functional aspects of systems.
- Behavioural specification, i.e. describing how things change is the strong side of Z.
- The Z notation is used for the formal definition of all main control-flow testing criteria. These definitions help to eliminate the possibility of inaccuracy in understanding, which is generally happens with definitions in natural language.
- It represents both static and dynamic aspects of a system.

The major weaknesses of Z method are as follows:

- Specification written using Z notation cannot be used to generate computer source code directly.
- The weak side of Z notation is a lack of graphical notation.
- This method does not have the facility of exception handling.
- It does not provide support for concurrency control.
- Z does not support all aspects of design.
- Z formal specification is non- executable i.e. it does not contain information on how specified functionality should be achieved.

Strengths and Weaknesses

- The major strengths of VDM method are as follows:
- Specification written using VDM can be used to generate computer source code directly.
- Specification is comprehensive and precise thus, easy to understand.
- VDM emphasizes on the feature of concurrency control.
- Free tools support with reference to VDM is available in the market.
- VDM provides the equivalence of programming language concepts as part of compiler correctness arguments.
- VDM has the facility of explicit exception handling.

The major weaknesses of VDM method are as follows:

- VDM method does not support all aspects of designing of the system.
- It cannot specify usability, reliability, safety and performance requirements.
- VDM Tools has lack of usability. There is no internal editor for the models. In this case the user always has to use the editor, for example Microsoft Word to change the specification.
- In VDM Tools, the error list cannot be emptied and so it is hard to see which errors are new and which have already been fixed.

Strengths and Weaknesses

The major strengths of B method are as follows:

- Specification written using B can be used to generate computer source code directly.
- Free tools support is available in the market.
- B models the system in an abstract machine notation that can helps to design system generate its code.
- In this method system design is refined and tested efficiently.
- B method is a tool-supported formal methods. It is based on AMN (Abstract Machine Notation) which is used in the development of correct software.

The major weaknesses of B method are as follows:

- It does not provide support for concurrency control.
- B is slightly more low-level.
- It is more focused on refinement to code rather than just formal specification.
- No support for object oriented concept is provided by B-method.

Strengths and Weaknesses

The major strengths of OBJ method are as follows:

- Formal specification languages become more attractive and easy to use by using object oriented concepts.
- Object orientation fills the gap between structural (static) and behavioral (dynamic) specifications.
- Inheritance helps in reusability of specification.
- Class specialization establishes refinement relations.
- Object orientation not only helps in the formal reasoning stages but also simplifies the actual writing of specifications.
- An object-oriented formal method can be integrated smoothly in the whole development cycle.

The major weaknesses of OBJ method are as follows:

- The adoption of the object paradigm brings powerful abstraction and structuring mechanisms but at the same time creates some additional problems for example, if we want to use the constructs such as classes, inheritance and objects in a truly formal methodology they need to be formally defined and integrated within the semantic model of the language.
- The number of interpretations and contradictory terminology makes it difficult to integrate or even compare different approaches. Attempts to introduce formalism into the paradigm have often been hampered by these terms and concepts.

Strengths and Weaknesses

The major strengths of LARCH are as follows:

- It is used to specify interfaces between programs in different languages.
- Larch family is able to cope with the features of different implementation languages.
- It supports the feature of abstraction.
- Larch interface languages encourage a style of programming that emphasizes the use of abstractions, and each provides a mechanism for specifying abstract types.
- To make it possible to validate specifications before implementing or executing them, Larch permits specifiers to make assertions about specifications.

Weaknesses

- Some larch specifications can be executed; most of them cannot.
- Larch style of specification emphasizes simplicity and clarity rather than executability.
- It is not focused to the major development phase i.e. requirement phase.

Strengths and Weaknesses

The major strengths of CSP are as follows:

- CSP is focused to verify safety and liveness properties of the system.
- CSP allows the description of systems in terms of component processes that operate independently, and interact with each other through message-passing communication.
- Main focus of CSP is in software design and it is usually used to design safety-critical systems.
- CSP is well-suited to modeling and analyzing systems that incorporate complex message exchanges, it has also been applied to the verification of communications and security protocols.
- There exist a number of tools for analyzing and understanding systems described using CSP.

The major weaknesses of CSP are as follows:

- CSP is well-suited to design software; it is not focus at requirements specification phase.
- Working of CSP is very difficult to understand.
- It is only used in concurrent system.
- It needs at least two processes i.e. sending and receiving process for communication.
- A subordinate process needs to know the name of its using process; this complicates construction of libraries of subordinate processes.

Comparison of Formal Methods and Future Benefits

Attribute	Z-Method	VDM	B-Method	OBJ	Larch	CSP
Concurrency Control	×	√	×	√	×	√
Supporting Tools	√	√	√	√	√	√
Support for abstraction	√	√	√	√	√	√
Object Oriented Concept	√	√	×	√	√	√
Structuring	√	×	√	√	√	√
Requirements Phase Perspective	√	√	×	×	√	×

Concurrency Control: Concurrency is a property used in distributed system that enables software systems to be served in large-scale distributed systems. This property allows several computations to execute simultaneously, and potentially interact with each other.

Supporting Tools: It helps in automation of any process. Supporting tools makes the steps easier; therefore, tools support is highly recommended.

Support for Abstraction: Abstraction is the process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details. **Abstraction** captures only essential details about an object that are relevant to the current perspective.

Object Oriented Concept: The object oriented concepts such as inheritance, polymorphism, and encapsulation are supported by some formal specification languages. Object oriented programming is an approach for developing software system based on the concepts of classes and objects.

Structuring: It is a mechanism for combining specifications, for example, to handle error handling or status information.

Requirements Phase Perspective: Requirements phase is the backbone of any software to be developed [18]. As it is well accepted by the research community, it is necessary for any methodology to consider this perspective.

Formal Languages in SDLC

1. **Specification (Requirements Analysis Phase)**

- Specification is the major step in development life-cycle.
- It is the process of describing a system behavior and its desired properties.
- Formal specification languages describe system properties that might include functional behavior, timing, behavior, performance characteristics and internal structure.
- For specifying the behavior of **sequential systems** formal methods such as Z, VDM and Larch are used.
- Formal methods such as CSP, CCS, State charts, Temporal Logic, Lamport and I/O automata, focus on specifying the **behavior of concurrent systems.**

Formal Languages in SDLC

2. Verification (Testing Phase).

- **Verification** is the process to prove or disprove the correctness of a system with respect to the formal specification or property.
- For the **verification** of the code, there are two important forms: **Model Checking** and **Theorem proving**.
- In **model checking**, a finite state model of the system is build and its state space is mechanically investigated. Two well known and equivalent model checkers are **NuSMV** and **SPIN**.

Outline

- SDLC
- Logic and Testing
  - Test Plan and Test case
  - Test Flow
  - Test Size
  - Test Depth
- Logic and Testing
- Other Testing
  - Unit Testing
  - Control Flow Testing
  - Data Flow Testing
  - Domain Testing

SDLC Revision

- The SDLC can be divided into seven (7) stages
  - Initial Study → Team collects information regarding the problem.
  - Analysis → Team convers collected information in to high level specifications
  - Design → Team creates low level specifications for the solution.
  - Development → built the solution based on the given specification.
  - Testing → verify the validity of the software specifications.
  - Implementation → Move the software to production.
  - Review → Team and client review the software.

Testing Concept

Test Flow

- There are two Test Flows

1. **Top down** : a test flow that starts from a general level down to the specific detail.
   - Example an **inventory system** that start from a inventory module down to sub module called **product** module.
2. **Bottom Up** : Starts at specific and move to a general level.
   - Example an inventory system that start from a **product** module up to **Inventory** module called

Testing Concept

Test Size: Based on the test coverage we have 4 types of test sizes

1. Unit Testing refers to testing specific module independently. Testing **customer** module, and then testing **product** module alone.

2. Integration Testing refers to a test that joins individual modules. Example for an inventory system testing the product and customer module to test out the **sales invoice** module.

3. System Testing : This is a test that starts to studies the system environment of the software. Example for an inventory system will be to test if the bar code reader at the POS can read the barcode label on the product.

4. Acceptance Testing: is **a quality assurance (QA) process that determines to what degree an application meets end users' approval**. Depending on the organization, acceptance testing might take the form of beta testing, application testing, field testing or end-user testing.

Testing Concept

- Based on the **Test Depth** we have

1. **Black box testing** Due to its nature, black box testing is sometimes called specification-based testing, closed box testing, or opaque box testing. focuses on understanding user experience, which means testers do not require in-depth technical knowledge to carry it out.
2. **White box testing** also called code-based testing, glass box testing, open box testing, clear box testing, and transparent box testing.
   - The software application's internal coding, design, and structure are examined in white box testing to verify data flow from input to output.
   - White box testing is leveraged to improve design, usability, and application security.

Testing Concept

3. Gray box testing is a software testing method to test the software application with partial knowledge of the internal working structure. It is a **combination of black box and white box testing** because it involves access to internal coding to design test cases as white box testing and testing practices are done at functionality level as black box testing.

To what extent do we test ?

- Each line of code ? How many line of code do we have ? Feasible ?
- Each module ? Input/output ? Functions ? Decisions ?
- Each Component ? All paths within ? One path ?
- The system at once ? Business logic ? Business specification ? Security ? Safety ?

Logic and Testing

Covering **Logical Expressions** in software testing

Logic Tests are intended to choose some subset of the total number of truth assignments to the expressions in specification or code.

Logic Predicates and Clauses

A predicate is an expression that evaluates to a boolean value – True or False

- Predicates can contain
  - boolean variables
  - non-boolean variables that contain
    >, =, <=, !=
  - boolean function calls

- Sources of predicates
  - Decisions in programs
  - Guards in finite state machines
  - Decisions in UML activity, state chart, work flow and etc
  - Requirements, both formal and informal
  - SQL queries

Logic and Testing

- We use predicates in testing as follows :
  - Developing a model of the software as one or more predicates
  - Requiring tests to satisfy some combination of clauses
- Given the following abbreviations:
  - *P* is the set of predicates
  - *p* is a single predicate in *P*
  - *C* is the set of clauses in *P*
  - C_p is the set of clauses in predicate *p*
  - *c* is a single clause in *C*

Predicate and Clause Coverage in testing

Predicate Coverage (PC) : For each *p* in *P*, *Test Rule (TR)* contains two requirements: when *p* evaluates to true, and when *p* evaluates to false.

Logic and Testing

Predicate Coverage Example

((a < b) ∨ D) ∧ (m >= n*o)

Predicate = true

a = 5, b = 10, D = true, m = 1, n = 1, o = 1

= (5 < 10) ∨ true ∧ (1 >= 1*1)

= true ∨ true ∧ TRUE

= true

When p evaluates to true.

Predicate = false

a = 10, b = 5, D = false, m = 1, n = 1, o = 1

= (10 < 5) ∨ false ∧ (1 >= 1*1)

= false ∨ false ∧ TRUE

= false

When p evaluates to false.

Logic and Testing

```
if ((gear == 5 || gear == 6) && speed >= 100)
        print("highway")
else
        print("some other road");
```

P is tested for input combinations

P : (g = 5 $\vee$ g = 6) Λ s >= 100

given g=5, g=5, s==105

true Λ true=TRUE

***p* evaluates to true.**

P : (g = 5 $\vee$ g = 6) Λ s >= 100

given g=5, g=5, s==80

true Λ false =TRUE

***p* evaluates to false.**

Logic and Testing

Consider a safety critical system with a **valve** that might be either **open** or **closed**, and that has several modes, two of which are "**Standby**" and "**Operational**."

Suppose that there are two constraints:

1. The valve must be **open** in "**Operational**" and **closed** in all other modes.
2. The mode cannot be both "**Standby**" and "**Operational**" at the same time.

These constraints lead to the following clause definitions:

a = "The valve is closed"

b = "The system status is Operational"

c = "The system status is Standby"

The two constraints can now be formalized as:

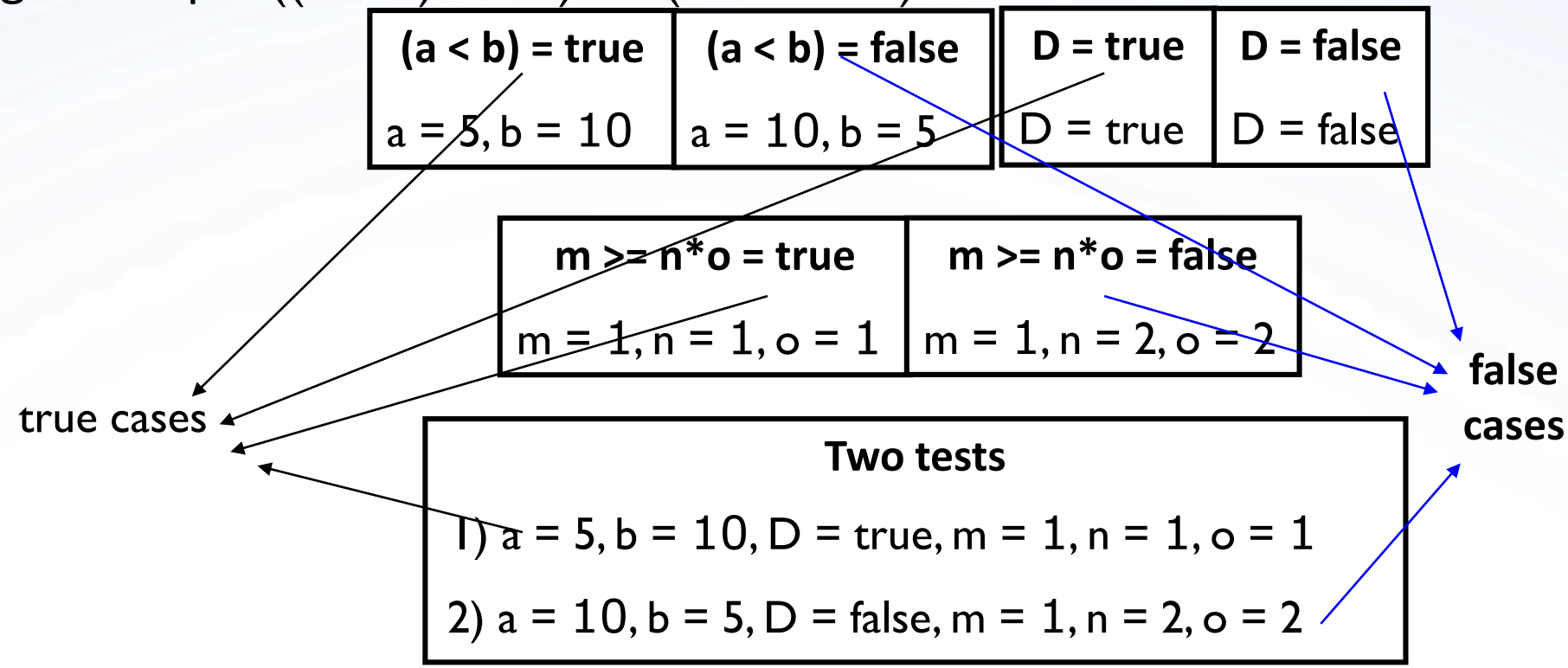
1. $\neg a \leftrightarrow b$
2. $\neg(b \wedge c)$

Develop truth assignment table for this logic

Logic and Testing

Clause Coverage (CC) : For each *c* in *C*, *TR* contains two requirements: when *c* evaluates to true, and when *c* evaluates to false.

Clause Coverage Example ((a < b) ∨ D) ∧ (m >= n*o)



Logic and Testing

- PC does not fully exercise all the clauses, especially in the presence of short circuit (skipping branch in codes) evaluation
- CC does not always ensure PC
  - That is, we can satisfy CC without causing the predicate to be both true and false
- The simplest solution is to test all combinations …
- **Combinatorial Coverage**
- **Combinatorial Coverage (CoC) : For each p in P, TR has test requirements for the clauses in CP to evaluate to each possible combination of truth values.**

Logic and Testing

- CoC requires every possible combination
- Sometimes called Multiple Condition Coverage

	a < b	D	m >= n*o	((a < b) ∨ D) ∧ (m >= n*o)
1	T	T	T	T*
2	T	T	F	F
3	T	F	T	T*
4	T	F	F	F
5	F	T	T	T*
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

There are three cases the system meets system requirements as in T*

- But quite expensive!

2^N tests, where N is the number of clauses

Impractical for predicates with more than 3 or 4 clauses

- Control software often has many complicated predicates, with lots of clauses

Logic Expressions from Source

- Predicates are derived from decision statements
  - –if, while, for, switch, do-while
- In programs, most predicates have less than four clauses
  - –In fact, most have just one clause
- *Reachability* : Each test must reach the decision
- *Controllability* : Each test must cause the decision to have specific truth assignment
- *Internal variables :* Predicates variables that are not inputs

How to cover the executions

```
IF (A>1) & & (B=0) THEN X=X/A
 END
IF (A=2) | | (X>1) THEN X=X+1
END
```

- Choose values for A,B,X.
- Value of X may change, depending on A,B.
- What do we want to cover?

  Paths? If or then branch

  Statements? X=X/A

  Conditions? A>1

How to cover the executions

1, Execute every statement at least once

for **A=2,B=0,X=3**

```
IF (A>1) & & (B=0) THEN X=X/A
END
IF (A=2) | | (X>1) THEN X=X+1
END
```

2, Decision coverage (if, while checks)

Each decision has a true and false outcome at least once.

Can be achieved using

- **A=3,B=0,X=3, and A=2,B=1,X=1**

Problem: Does not test individual *conditions*. E.g., when **X>1** is erroneous in second decision.

How to cover the executions

3, Condition coverage

Example 1:

A=1,B=0,X=3

A=2,B=1,X=0

```
IF (A>1) & & (B=0) THEN X=X/A          END
IF (A=2) | | (X>1) THEN X=X+1          END
```

Problem: covers only the path where the first test fails and the second succeeds.

Example 2:

A=2,B=1,X=0

```
IF (A>1) & & (B=0) THEN X=X/A      END
IF (A=2) | | (X>1) THEN X=X+1      END
```

Did not check the first **THEN** part at all

Can use condition + decision coverage.

How to cover the executions

4, Multiple Condition Coverage

Test all combinations of all conditions in each test.

- **A>1,B=0**
- **A>1,B≠0**
- **A≤1,B=0**
- **A≤1,B≠0**
- **A=2,X>1**
- **A=2,X≤1**
- **A≠2,X>1**
- **A≠2,X≤1**

```
IF (A>1) & & (B=0) THEN X=X/A
   END
IF (A=2) | | (X>1) THEN X=X+1
   END
```

**Further optimization: not all combinations, but given conditions C and D,

For C /\ D, check (C, D), (¬C, D), (C, ¬D).

For C \/ D, check (¬C, ¬D), (¬C, D), (C, ¬D).

Example Applications

Given the following business specification for an ATM application

Actors

Customer

Bank Server

Business Functions

Card Reader checks ATM card

PIN Entry Screen [Customer enters the PIN]

Account Options Screen [Customer gets main screen]

Exceptions Conditions

Unreadable Card Screen

PIN Error Screen or No PIN Screen

Account Locked Screen

Develop the possible predicates and clauses from the above business description. Design exhaustive [predicate and clause] test coverages