Formal Methods in Software Engineering

Credit Hours: 3+0

Dr.Wafa Basit

Software life cycle (SLC)

- A *software life cycle (SLC)* is a structure describing development, deployment, maintenance, and dismantling of a software product.
- There are several models for such processes, describing different approaches on how to develop software (Waterfall, Spiral, Iterative, Agile development, etc.).
- The main objective of this course is to discuss the use of Formal Methods in the software development process, independently of the model used.

What are Formal Methods?

- Formal Methods are centred around a notation known as a formal specification language.

  Formal Semantics

  - Mathematical base allows precise notions.
  - Unambiguous.
  - Allows consistency, correctness, specification and implementation to be expressed.

  Increase Human Understanding of Specified System.

  &

  Allow the possibility of formal reasoning and development.

What Are Formal Methods

- Formal methods refer to a variety of mathematical modeling techniques that are applicable to computer system design.
- They include activities such as:
- - System specification
- - Specification analysis and proof
- - Transformational development
- - Program verification

Definition

- “Formal methods are mathematical approaches to software and system development which support the rigorous specification, design, and verification of computer systems.” [Fme04]

- “[They] exploit the power of mathematical notation and mathematical proofs.” [Gla04]

Formal Methods

- Formal Methods can play multiple roles in the software design process
  - Ensure that a computer system meets its requirements..
  - Some software development standards actually require the use of Formal Methods for high integrity levels.
  - Mostly, Formal Methods help to make system descriptions precise and to support system analysis.
  - However, their application is feasible only when they are supported by tools.

History

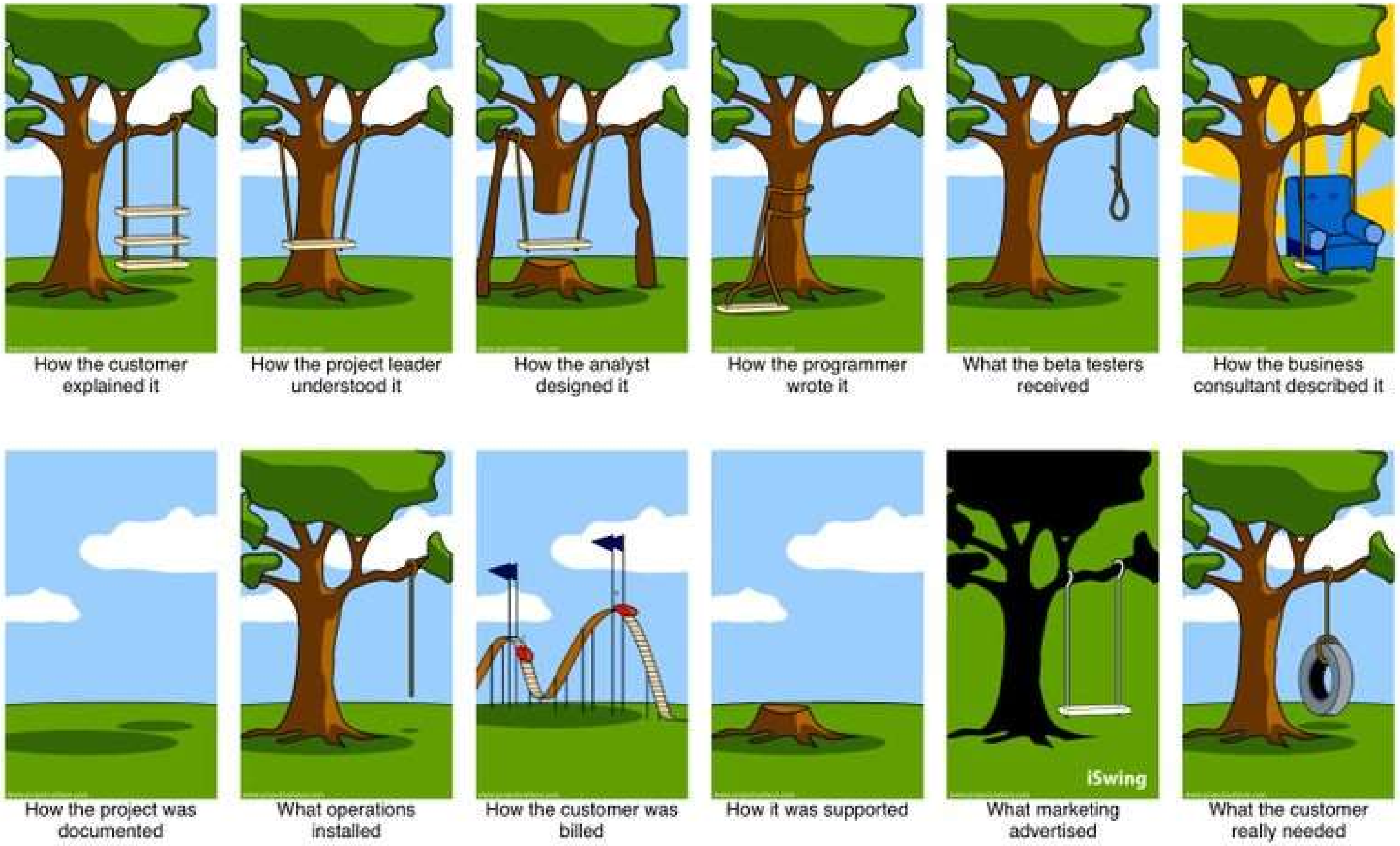
- - 1940's: Turing annotated program states for logical analysis of sequential programs.
- - 1960's: Floyd, Hoare, and Naur recommended axiomatic techniques to prove program specifications.
- - 1970's: Dijkstra used formal calculus for non-deterministic programs.
- - Interest in formal methods in software engineering has continued to grow.

Why formal methods?

- Informal methods are open to interpretation and ambiguity, and often incomplete and inconsistent.
- Formal methods have some basis in mathematics and allow us to reason about the system we are building.
- They help us accurately capture our intent so that it is precise and clear to others.

How Projects Really Work (version 1.5)

Create your own cartoon at www.projectcartoon.com


How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
What the beta testers received
How the business consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
iSwing
What marketing advertised
What the customer really needed

Why formal methods?

- There are lots of different formal methods.
- Like programming languages, each formal methods has strengths and weaknesses.
- Like programming languages, some formal methods are tailored to a specific problem domain, e.g., real-time systems requiring a discussion of temporal semantics.
- Formal methods are used to specify aspects of the software system – behavior, data relationship, temporal semantics

Why aren’t they used everywhere?

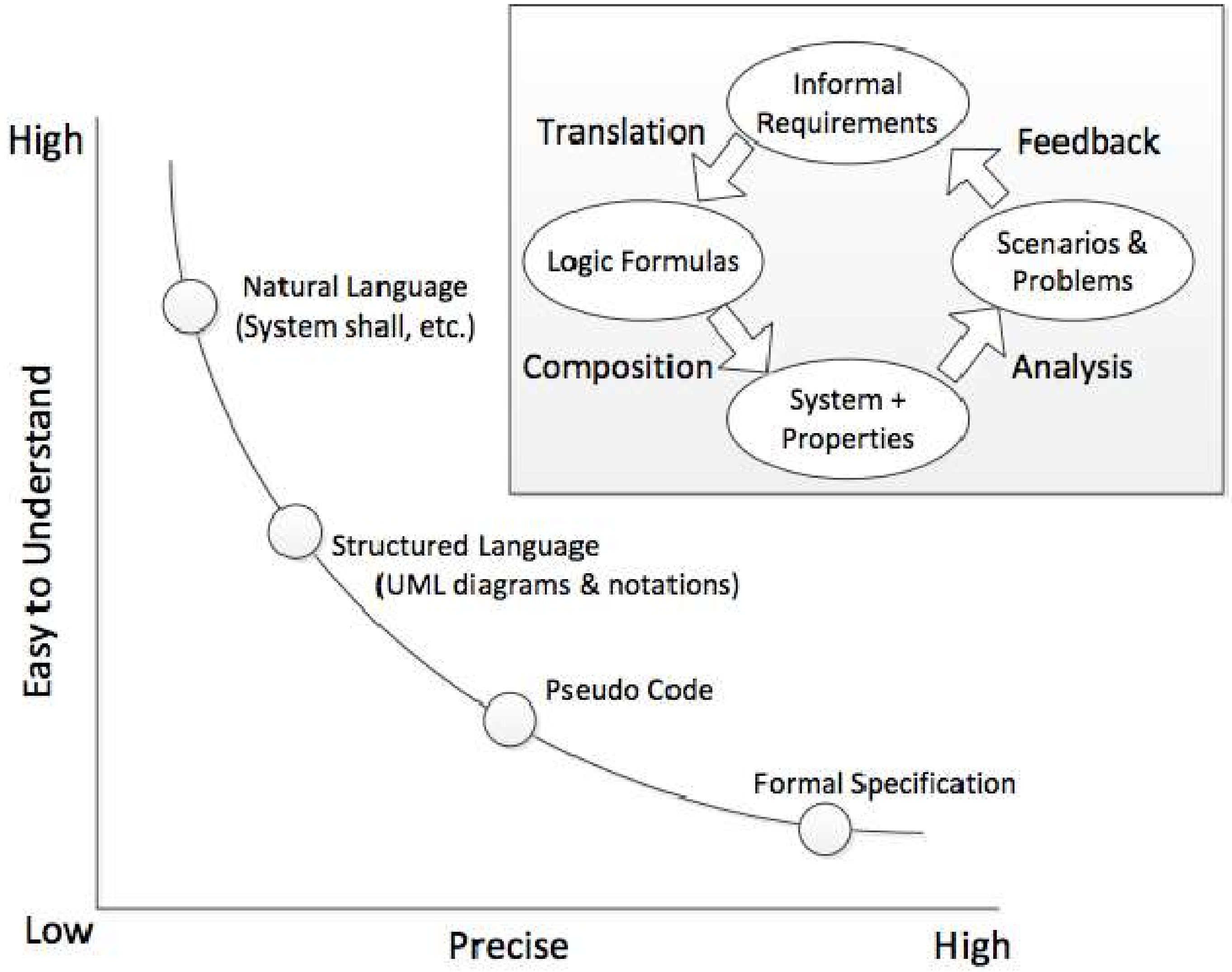
- *Formal methods are controversial. Their advocates claim that they can revolutionize [software] development.*
- *Their detractors think that they are impossibly difficult. Meanwhile, for most people, formal methods are so unfamiliar that it is difficult to judge the competing claims.* – Anthony Hall, 1990.
- We will gain an appreciation of formal methods over the next few lectures to understand their impact on software engineering in the years to come.

Why Aren’t Formal Methods Widely Used?

- Software quality improvements reduce perceived need.
- Time-to-market pressures.
- Limited scalability.
- Higher cost
- Lack of expertise
- Lack of tool support

Seven Myths of Formal Methods

- - Formal methods can guarantee that software is perfect.
- - Work by proving that programs are correct.
- - Only highly critical systems benefit from their use.
- - They involve complex math.
- - They increase the cost of development.
- - They are incomprehensible to clients.
- - Nobody uses them for real projects.


High
Easy to Understand
Low
Precise
High
Natural Language
(System shall, etc.)
Structured Language
(UML diagrams & notations)
Pseudo Code
Formal Specification
Informal Requirements
Translation
Feedback
Logic Formulas
Scenarios & Problems
Composition
Analysis
System + Properties

Formal methods - limitations

- Formal methods have not become mainstream software development techniques as once predicted.
  - Other SE techniques have been successful at increasing software quality, so the need to formalism has decreased.
  - Economic pressures make time-to-market a key driver, rather than a low error count. Formal methods do not reduce time to market.
  - Formal methods are had to scale to large software systems.

Formal methods - limitations

  - Scope of formal methods is limited – not well suited to specifying and analyzing user interfaces and user interactions.
- Formal methods have limited practical applicability.
- Principle benefit is reduction in number of errors. Main area of applicability is critical systems.

Deficiencies of informal methods

- Natural language (English) is an example of an informal method.
  - It is often ambiguous, incomplete, insufficient, and contains contradictions and mixed levels of abstractions.
- We often try to address these deficiencies by supplementing the natural language description with diagrams.
- These diagrams often lack structure and have similar deficiencies to natural language.
- Coupled with sloppiness in analysis and design, poor attention to detail and the lack of thorough reviews, the use of informal methods can lead to a software disaster.

Problems

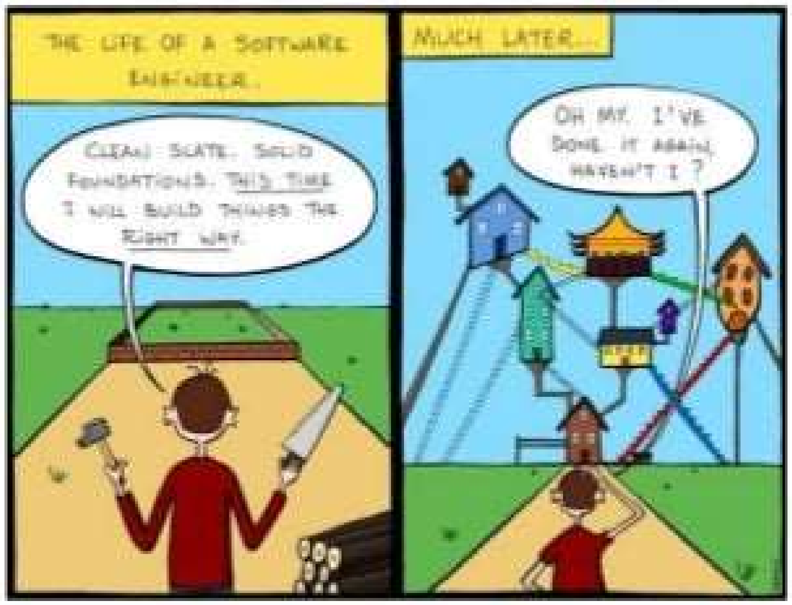
- *Contradiction* – sets of statements that are in variance with each other. They are often written by different people and separated by many pages in the specification. They are therefore difficult to detect.

Problems

- *Ambiguity* – statements that can be interpreted in a number of ways. They usually do not appear ambiguous to the author.
  - *The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited into the login file when an operator logs into the system.*
  - What does "it" refer to – the operator or the password?

Problems

- *Vagueness* – often occurs because the spoecification document is bulky. Achieving a high level of precision consistently is an almost impossible task.
  - *The interface to the system used by the radar operators should be user-friendly.*
  - The above statement contains no useful information!


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

Problem

- *Incompleteness* – missing details such as a failure to specify what should occur in certain conditions.
  - *The system should maintain the hourly level of the reservoir from depth sensors in the reservoir. These values should be stored for the past six months.*
  - *The function of the AVERAGE command is to display on a PC the average water depth for a particular sensor between two times.*
  - What happens if the user specifies a period greater than six months?

Problems

- *Mixed levels of abstraction* – leads to difficulty in seeing the overall functional architecture of a system.
  - For example, if the following statements were intermixed:
    - *The purpose of the system is to track the stock in a warehouse.*
    - *When the loading clerk types in the* withdraw *command he or she will communicate the order number, the identity of the item to be removed, and the quantity removed.The system will respond with a confirmation that the removal is allowable.*
  - This leads to a lack of clarity.

Formal Methods

- Formal Methods are often used in safety-critical areas, where human life or health or a large sum of money depends on the correctness of software.
- Traditional testing catches bugs, but it cannot prove the absence of errors. Formal methods, on the other hand, provide a mathematical foundation to verify system properties, ensuring that all possible scenarios have been considered.
- This can drastically reduce bugs, improve security, and enhance system stability.

Use in the Design Process

- Specification

  Description of system to be developed at any level of detail desired.

  Formal specification can be used:

  1. to guide further development.
  2. verify requirements of system are completely and accurately specified.

Definition of Formal Specification

- A formal specification consists of three components:
- 1. Syntax - Grammatical rules to determine if sentences are well formed.
- 2. Semantics - Rules for interpreting sentences meaningfully within the domain.
- 3. Proof Theory - Rules for inferring useful information from the specification.

Types of Formal Methods

- - Abstract State Machines/Finite State Machines
- - B-Method
- - Z (Specification Language)
- - Unified Modeling Language (UML)
- Object Constraint Language (OCL)
- - Others: CommUnity, Estelle, Esterel, Lotos, Overture Modeling Language, Petri Nets, etc.

Predicate Logic – Definition

- A predicate is an expression of one or more variables defined on some specific domain. A predicate with variables can be made a proposition by either assigning a value to the variable or by quantifying the variable.
- The following are some examples of predicates −
- Let E(x, y) denote "x = y"
- Let X(a, b, c) denote "a + b + c = 0"
- Let M(x, y) denote "x is married to y"

Predicate Calculus

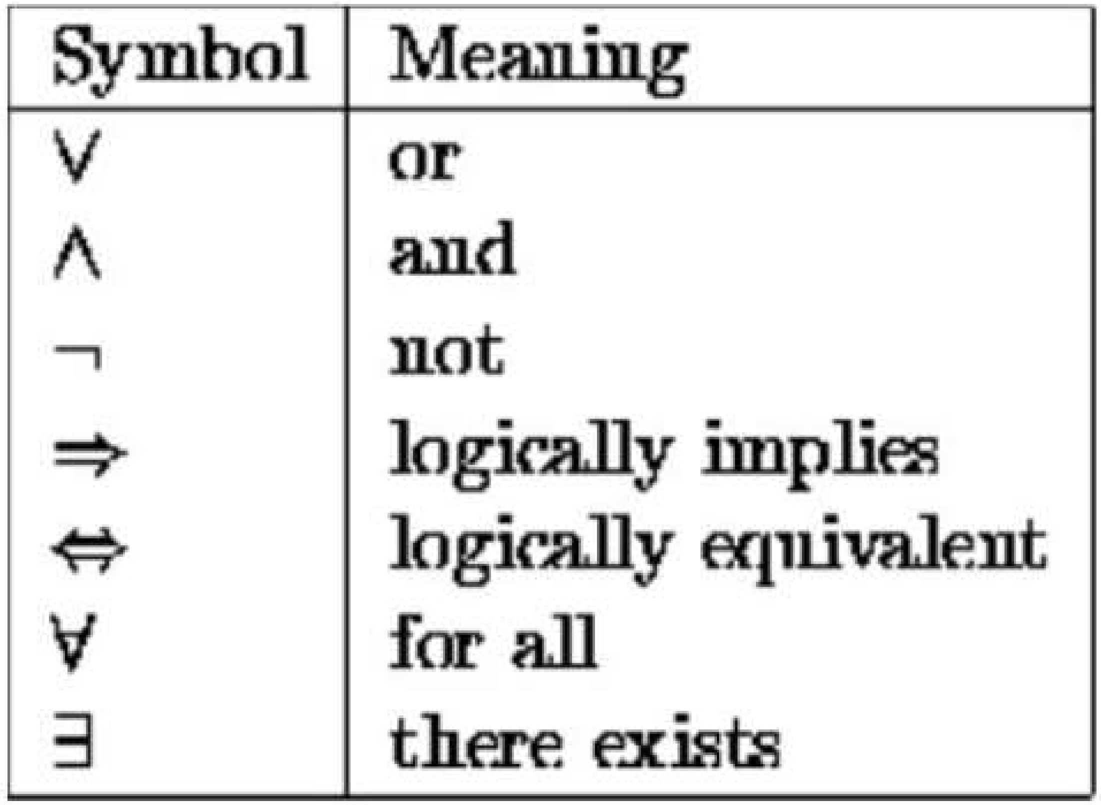
- - A formal language for expressing propositions.
- - Components:
  - - Constant
  - - Variable
  - - Predicate
  - - Function
  - - Connective
  - - Quantifier

Predicate Calculus

1. Whoever can read is literate.
2. Dogs are not literate.
3. Some dogs are intelligent.
4. Some who are intelligent cannot read.

1. $\forall x\ [R(x) \Rightarrow L(x)]$
2. $\forall x\ [D(x) \Rightarrow \neg R(x)]$
3. $\exists x\ [D(x) \wedge I(x)]$
4. $\exists x\ [I(x) \wedge \neg R(x)]$

Predicate Calculus



Symbol	Meaning
$\vee$	or
$\wedge$	and
$\neg$	not
$\Rightarrow$	logically implies
$\Leftrightarrow$	logically equivalent
$\forall$	for all
$\exists$	there exists

Levels of Rigor

- Specifications, models, and verifications may be done using a variety of techniques.
- *Level 1* represents the use of mathematical logic to specify the system.
- *Level 2* uses pencil-and-paper proofs.
- *Level 3* is the most rigorous application of formal methods.

Do We Really Need Formal Methods?

- Design errors can lead to catastrophic failures, including:
- - Physical failure
- - Human error
- - Environmental factors
- - Design errors (major culprit)
- Examples:
- - Therac-25 radiation overdose killed two people
- - Titan I cost taxpayers $1.23 billion due to software malfunctions.

Effects of Design Errors

- Denver Airport's computerized baggage handling system delayed opening by 16 months. Airport cost was $3.2 billion over budget.
- NASA's Checkout Launch and Control System (CLCS) cancelled 9/2002 after spending over $300 million.

The Promise of Formal Methods

- Formal methods can:
- - Improve software quality
- - Reduce cost of verification
- - Improve development process rigor
- - Reduce specification errors
- - Provide a basis for test data

Success of Formal Methods

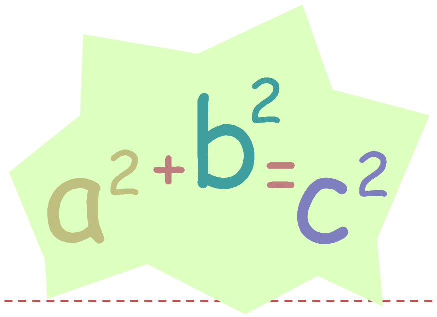
- Examples of successful applications:
- - Transportation systems
- - Information systems
- - Telecommunication systems
- - Power plant control
- - Security systems

Formal languages

It is desirable to use a specification notation with a **fixed**, **unambiguous**, **semantics**.

Notations that have a fixed semantics are known as **formal notations**, or **formal languages**.

A fixed semantics is achieved by defining a language in a completely unambiguous way using a mathematical framework.



Inconsistent specifications

A specification is **inconsistent** when it contains within it contradictions.

Withdraw:

"Receives a requested amount to withdraw from the bank account and, if there are sufficient funds in the account, meets the request.

Returns a boolean value indicating success or failure of the attempt to withdraw money from the account."

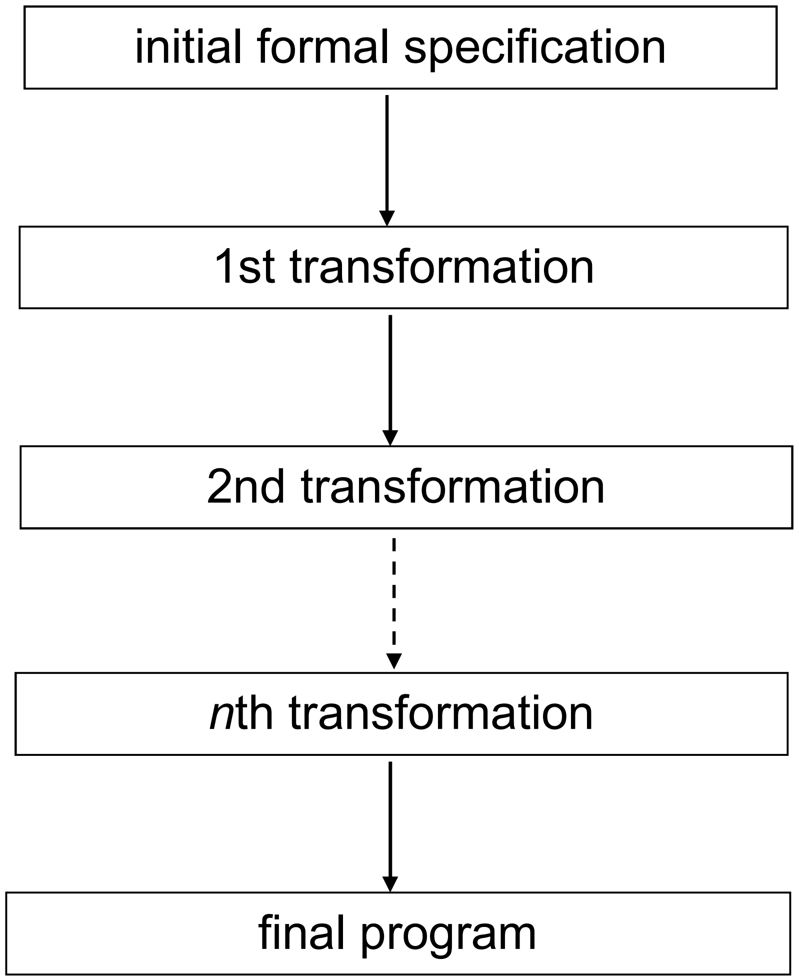
OVERDRAFT?

Elevator problem

- Let us define the elevator problem as:
  - We need to move n elevators between m floors.

1. Each elevator has a set of m buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the elevator
2. Each floor, except the first floor and the top floor has two buttons, one to request an up-elevator, and one to request a down-elevator. These buttons illuminate when pressed. The illumination is cancelled when an elevator visits the floor and then moves in the desired direction.
3. When an elevator has no requests, it remains at tis current floor with its doors closed.

Formal Methods



A formal method includes a **proof system** for demonstrating that each transformation preserves the formal meaning captured in the previous step.

Advantages of formal methods

- formal specifications can help considerably in generating suitable test cases;
- the discipline required in producing a formal specification allows for feedback on system specifications at early development stages, increasing confidence that the specification accurately captures the real system requirements;
- important properties of the initial specification can be checked mathematically and incorporated as run-time checks in the final program;
- proofs can help uncover design errors as soon as they are made, rather than having to wait for testing of the final implementation;
- a proof of program correctness can be constructed that is a much more robust method of achieving program correctness than is testing alone.

Critical Analysis

1. Full formalisation considered:

   too difficult

   too time-consuming

   too expensive

   given

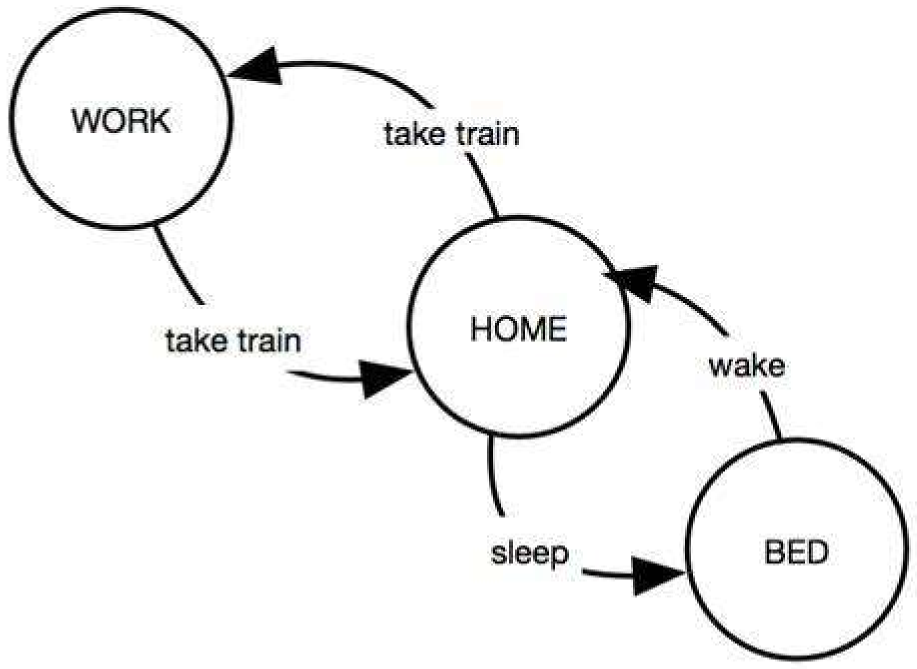
   1. *Expressiveness of languages involved*
   2. *Complexity of systems to be modelled.*

2. A good Human-Directed Proof requires high level of mathematical sophistication and expertise.

3. Automated-proof requires “guidance”.

Advantages of Formal Methods

- Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the observable properties and behaviors of these objects.
- There are several advantages to using formal methods for the specification and analysis of real-time systems.
  - the early discovery of ambiguities, inconsistencies and incompleteness in informal requirements
  - the automatic or machine-assisted analysis of the correctness of specifications with respect to requirements
  - the evaluation of design alternatives without expensive prototyping


WORK
take train
HOME
take train
wake
sleep
BED

Finite state machines

- In each of the n elevators, there are m buttons – one for each floor. These are called elevator buttons (EB).
- One each floor there are two buttons to request an up and down elevator. These are called floor buttons (FB).
- Let EB(e, f) denote the button in elevator e that is pressed to request floor f.
- EB(e, f) can be in two states:
  - EBON(e, f): Elevator Button (e, f) ON.
  - EBOFF(e, f): Elevator Button (e, f) OFF.

Finite state machines

- If the button is on and the elevator arrives at floor f, then the button is turned off. If the button is off, and it is pressed, then the button becomes On.
- These two events are:
  - EBP(e, f): Elevator Button (e, f) Pressed.
  - EAF(e, f): Elevator e Arrives at Floor f.

Finite state machines

$$\begin{tikzcd}\boxed{\text{EBOFF}(e, f)} \arrow[r, shift left, "\text{EBP}(e, f)"] & \boxed{\text{EBON}(e, f)} \arrow[l, shift left, "\text{EAF}(e, f)"]\end{tikzcd}$$

- The state transition diagram for an elevator button is given above.
- We introduce a predicate:
  - V(e, f): Elevator e is Visiting floor f.
- A predicate is a boolean function.
- If elevator button (e, f) is off (current state) and elevator button (e, f) is pressed (event) and elevator e is not visiting floor f (predicate), then the button is turned on.

$\text{EBOFF}(e, f) \land \text{EBP}(e, f) \land \neg V(e, f) \Rightarrow \text{EBON}(e, f)$

Finite state machines

- If the elevator is visiting floor f when button (e, f) is pressed, then nothing happens.
- If the elevator arrives at floor f and the button is on, then it is turned off.

  $\text{EBON}(e, f) \wedge \text{EAF}(e, f) \Rightarrow \text{EBOFF}(e, f)$
- Now we consider the floor buttons.
- FB(d, f) denotes the button on floor f that requests an elevator traveling in direction d.
- Each button has two states:
  - FBON(d, f): Floor Button (d, f) is ON.
  - FBOFF(d, f): Floor Button (d, f) is OFF.

Finite state machines

- If the button is on and an elevator arrives at floor f traveling in direction d, then the button is turned off.
- If the button is off, and it is pressed, then the button comes on.
- These events are:
  - FBP(d, f): Floor Button (d, f) Pressed
  - EAF(1..n, f): Elevator 1 or … or Elevator n Arrives at floor f.
- $$\begin{tikzcd} \boxed{\text{FBOFF(d, f)}} \arrow[r, shift left, "\text{FBP(d, f)}"] & \boxed{\text{FBON(d, f)}} \arrow[l, shift left, "\text{EAF(1..n, f)}"] \end{tikzcd}$$

Finite state machines

- We define predicate S as follows:

  S(d, e, f): Elevator e is visiting floor f and the direction in which it is about to move is either up (d = U), down (d = D), or no requests are pending (d = N)

- This predicate is actually a state.
- The transition rules for floor buttons is then:

  $\mathrm{FBOFF}(d, f) \wedge \mathrm{FBP}(d, f) \wedge \neg S(d, 1..n, f) \Rightarrow \mathrm{FBON}(d, f)$

  $\mathrm{FBON}(d, f) \wedge \mathrm{EAF}(1..n, f) \wedge S(d, 1..n, f) \Rightarrow \mathrm{FBOFF}(d, f), d = D \vee U$

- V(e, f) can be defined in terms of S(d, e, f)

  $V(e, f) = S(U, e, f) \vee S(D, e, f) \vee S(N, e, f)$