

1 Requirements

The process of **requirements analysis** and **system specification** involves examining the behaviour of the proposed system and asking questions which enable a **correct** specification to be developed.

- requirements analysis
- system specification
- understand behaviour
- ask questions
- enable correct specification to be developed
- think above code level
- start with specification then implementation
- specification drives design
- testing drives implementation

Key Questions: 1) Writing a spec to drive design. Using math to write formal spec. Then model checking. 2) Safety critical obligations.

2 Think Above Code Level by Leslie Lamport

1. We should think before we start writing any code.
2. If thinking without writing then you're only pretending to think.
3. What to write? Blueprints of Programs called specifications.
4. Specifications? Spectrum of blueprints (i.e. simple or complex, formal (mathematical) or informal (prose)).
5. The best type of specifications are in middle called Mathematical Prose.
6. Code for concurrent or distributed systems is going to be complex, subtle and critical.
7. Thus, there is a need for tools to check your blueprint.
8. Therefore, if you are going to use tools then you're going to need a formal language. This is because tools don't understand prose.
9. So how to write a Spec?
 - Writing requires thinking. So how to think about programs?

- You should think about programs like Scientists. Scientific thinking Make Mathematical Models of Reality.
 - In CS, reality consists of digital systems. processor chip or computer executing a program.
10. So, Models? Functions and Sequences of States are the two most useful basic models, according to Leslie Lamport.
 11. Function? we can model a program as a function that maps input output, or multiple inputs multiple outputs.
 12. In Math, function is a simply a set of ordered pairs.
 - $\langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 4 \rangle$
 - $\text{square}(2) = 4$
 - all the first elements of those pairs compose the domain of the function.
 13. To define a function, we specify it's domain:
 - Domain of square = NAT (0,1,2,3,4,5) vs NAT1 which excludes 0.
 - $\text{square}(x) = x^2$ for each x in its domain. (this is the definition!)
 14. Functions in math are simpler than functions in programming languages.
 15. The Functions Model's main limitation is it specifies what a program does. It does not specify how it does it.
 16. Furthermore, other limitations are:
 - how do we specify some programs that don't just map inputs to outputs.
 - how do we specify programs, like OS which we assume run forever (b/c it's convenient to think like this mathematically).
 - Thus, we use the Standard Behavioural Model Program execution is represented by a behaviour.
 - Behaviour is a sequence of states.
 - A state is an assignment of values to variables.
 - Thus, a Program is modelled as a set of behaviours. the behaviours that represent all possible executions of the program.
 17. So how do we Specify a set of behaviours?
 - by specifying a Safety Property and Liveness Property.
 - In practice, specifying safety just turns out to be more important (b/c that's where most errors are likely to occur).

18. How to Specify Safety Property?

- Two things:
 - (a) The set of all possible initial states.
 - (b) The next-state relation, describing all possible successor states of any state.
- So what Language should use to write these two things? we use MATH. i.e. as show in his Euclid's Algorithm example, Lamport generates two formulas- one for set of initial states and other for next-state relation.
- So how does this work? i.e. How do we get behaviours out of those formulas.

For Euclid's Algorithm

$$\text{Init: } (x = M) \wedge (y = N)$$

$$\begin{aligned} \text{Next: } & \quad \left(\begin{array}{l} \text{12} > \text{18} \\ \wedge \quad x' = \text{12} - \text{18} \\ \wedge \quad y' = \text{18} \end{array} \right) \quad \text{FALSE} \\ & \vee \left(\begin{array}{l} \text{18} > \text{12} \\ \wedge \quad y' = \text{18} - \text{12} \\ \wedge \quad x' = \text{12} \end{array} \right) \end{aligned}$$

Behavior:

$$[x = \text{12}, y = \text{18}] \rightarrow [x = \text{12}, y = \text{6}]$$

For Euclid's Algorithm

Init: $(x = M) \wedge (y = N)$

Next: $\begin{array}{l} \text{ ~~} (6 > 6 \\ \wedge x' = 6 - 6 \\ \wedge y' = 6 \text{)} \end{array}~~$ **FALSE**

$\vee \begin{array}{l} \text{ ~~} (6 > 6 \\ \wedge y' = 6 - 6 \\ \wedge x' = 6 \text{)} \end{array}~~$ **FALSE**

NO NEXT STATE

Behavior:

$[x = 12, y = 18] \rightarrow [x = 12, y = 6] \rightarrow [x = 6, y = 6]$

- - To Model non-determinism, use just have a next-state relation that allows multiple next-states for a current state. (there nothing magic or difficult about non-determinism)
19. What about Formal Specs? we need formal specs ONLY to apply tools.
 20. So we need a formal language called TLA+.
 21. You can model check TLA+ specs checks all possible execution of program, on a very small model
 22. It is extreamly EFFECTIVE and EASY to do.
 - You basically tell the model checker what the model is.
 - Models are instantiating value of constants. For eg, in Euclid's algo-rithm we'd have to tell checker what M and N are.
 - Benefit: You can write formal correctness proofs and check them mechanically in TLA+.
 - We do this by writing proofs in TLA+ then use theorm prover to check the proofs.
 - TLA+ is formal (language of mathematica) and PLUSCAL is puedocode. It looks like a programming language.
 23. Everyone thinks they are thinking but you're not writing down your thoughts, then you're fooling yourself.
 24. What programmer should know about thinking? you should think before you code write a spec before you code.

25. What code should you specify? any piece of code that someone else might want to use or modify.(eg: entire programmer system, class, method, or piece of code inside a method.)
26. What should you specify about the code? What it does, and how it does it. This called an Algorithm or high-level design.
27. How should you think about or specify your code? above the code level!- in terms of states and behaviours or functions. Mathematically/Rigorous.
28. Should be thinking Mathematically, even though, you're writing specs informally which pseudocode i.e. PlusCal.
29. How do you learn to write specs? by learning how to write formal specs this will help you write informal specs, which you will actually write.
30. You learn to write programs by writing them, running them and correcting your errors.
31. You can learn to write formal specs by writing them, running them with model check and correcting your errors.
32. TLA+ is a therefore a language for writing formal specification- it is great for **learning how to think mathematically**.
33. Writing Specs is hard b/c Thinking is hard. There is no royal road to Mathematics.