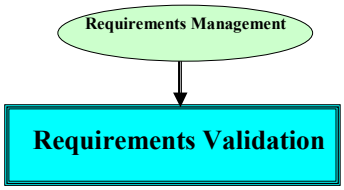
Requirements Validation



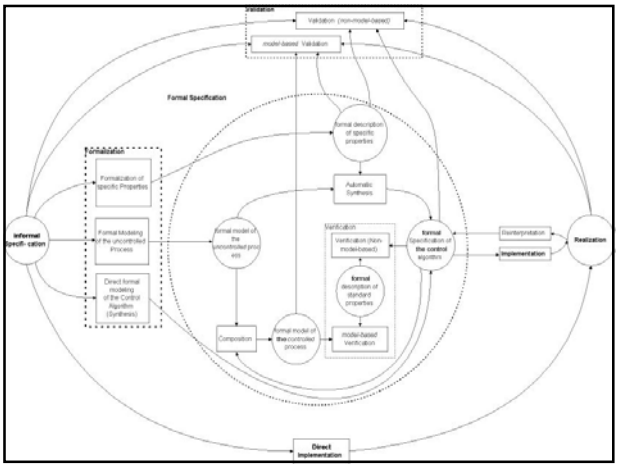
?? Validation, Verification, Accreditation !!

- Check if evrything is OK
  - With respect to what ?
  - Mesurement associated with requirements
- Dont get lost with terminology problem
- Some definitions
  - IEEE SRS
  - EIA-632

build the Right System
Build It Right

Content

- What the standards say
- Techniques and methods
- Well established techniques
- Case study



What the standards say (*)

- Main standards
  - IEEE P1220
  - EIA 632
  - DoD 2167A

Correctness and completeness

- **A correct, complete set of requirements is one that correctly and completely states the desires and needs of the sponsor.**
- **If the requirements are incorrect, the software may meet the requirements as stated, but will not do what the sponsor wants it to do.**
- **If the requirements are incomplete, the software may do only part of what the sponsor hoped it would do.**

Consistent

- **Consistency is obtained if the requirements do not contradict each other.**
- **Inconsistency results when one requirement contradicts another.**

Unambiguous

- **If a requirement is subject to more than one interpretation, it is ambiguous.**
- **Requirements should be stated simply and completely so that they are unambiguous .**

Functional

- **Requirements should state what the sponsor desires : the functions and activities to be performed by the system.**
- **They should not state how the problem is to be solved or what techniques are to be used.**
- **Such decisions should be left to the system designers.**

Verifiable

- **The requirements must be verifiable in two ways:**
  *** do the requirements satisfy the sponsor's needs ?**
  *** does the system satisfy the requirements?**
- **In the first case, the requirements must be compared to the sponsor's desires and needs. Do the requirements correctly and completely specify the sponsor's desires and needs?**
- **In the second case, once the system has been developed, it must be compared to the requirements. Does the system meet the requirements as they are stated?**

Traceability

- **Traceable and easily changed.**
- **The requirements should be organized and written in a segmented, top down manner that allows for easy use (traceability) and easy modification.**
- **A numbering system is useful to label the paragraphs and parts of the manual for cross referencing, indexing, and easy modification**

Techniques and methods

- Inspection
- Model Checking
- Simulation
- Prototyping
- Others

Inspection

- Most common simple et pragmatic method and can be
  - Manual : *Human sense principal instrument*
  - Automatic (CAI tool) : *for measurable issues*
- Most *evident* errors/faults can be detected

Inspection (example)

- This presentation shows how to carry out a one-half hour inspection with senior managers. The purpose is to show to make managers aware that they play a key-role in creating project delays by approving poor quality of requirements documents.
- The inspection results shown in this real-life example successfully predicted a project delay of 2 calendar years.
- Poor quality marketing requirements documents prove time and again to be a good predictor of project delays.
- The clue is that requirements documents with a high defect density are an indicator of a truly unprofessional engineering culture.

Inspection (example)

Rules

Introduce the following three rules for inspecting a requirements document:

Three Rules for Requirements:

- ***1. Unambiguous to intended Readership***
- ***2. Clear enough to test.***
- ***3. No Design (how to) mixed in***
  - with Requirements (how well)
  - MARK Design as "D"

Inspection (example)

Defect

Explain the definition of a Defect:

- A Defect is a violation of a Rule
- Note: If there are 10 ambiguous terms in a single requirement then there are 10 defects!

Inspection (example)

Severity

Explain:
• the definition of **Major** Defect
• the checkers must focus on finding Major Defects

- **Major:** a defect severity where there is potential of
  - high (x lost engineering hours) loss
  - later downstream (test, field).

Inspection (example)

Exit?

Agree with the management team on a numeric exit condition

- **Exit Conditions: (Requirements can go to Design, Test etc with little risk)**
  - ***Maximum 1 Major Defect/ (Logical) Page***

Logical Page *= 300 Non commentary words.*

Inspection (example)

The Job

- You have up to 30 minutes for checking One requirements Logical page from an 82 pages document
- Count all Rule Violations = Defects
- Classify Major and minor

Inspection (example)

- Page 81: 120 majors/p
- Page 82: 180 Majors/p
- Average 150 Majors/page x 82 page = 12,300 Majors in the document.

- If a Major has 1/3 chance of causing loss
- And each loss is avg 10 hours then total project Rework cost is about 41,000 hours loss.
- (This project was over a year late)
  - 1 year = 2,000 hour 10 people

Inspection (example)

Letter to Your Boss

Boss!
We have 2 options for the 82 page Requirements document.
Our sample shows that we have 180 Majors/Page.
We can spend 180 hours per page removing them with Inspection
We can rewrite the pages at a cost of 10hours each.
Or we can suffer 30% of these as bugs and fault, at an average removal cost of about 10 hours each (test and field debugging and re-testing), 1/3 of 180 x 10 = 600 hours per page if we do not rewrite (10 hours /Page) or remove before test (180 hours/Pages).

We suggest rewrite (changing something to avoid defect injection rate). But you have said you are against this. So we have to tell you that your option will delay our project by 600 hours x 82 = 49,200 hours.
Our project has 10 people on it, and they can do about 2,000 hours per year. So that is 20,000 work hours per year for our team. The approximate delay for your decision not to rewrite is this about 2.5 years worse Time To Market.
We will of course do what you say, but we wanted to be sure that you understood what your boss will blame you for later.
Your Loyal Servant, Tom

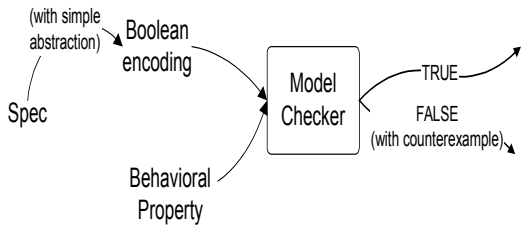
Simulation

- Abstract model of either requirements or the design solution
- Coverage of most scenarios

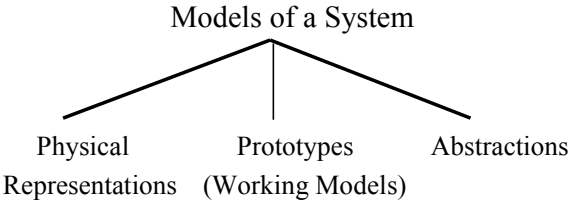
Model Checking

- Model checking is a method for formally verifying finite-state concurrent systems
- Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not
- Extremely large state-spaces can often be traversed in minutes
- The technique has been applied to several complex industrial systems
- Site : http://archive.comlab.ox.ac.uk/formal-methods.html

Model Checking and Formality



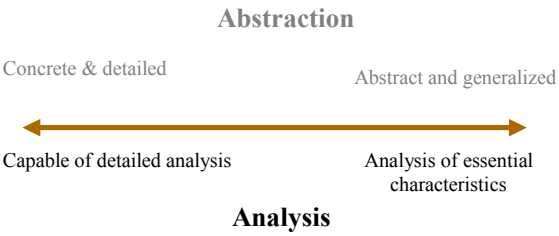
Model Checking and Formality



Model Checking and Formality

- A mathematical model is an abstract representation of a system employing mathematical entities and concepts
- Model should be expressive enough to capture the essential characteristics of the system being modeled
- If the model is intended for deductive reasoning about the underlying system, it should provide sufficient analytic power for this purpose

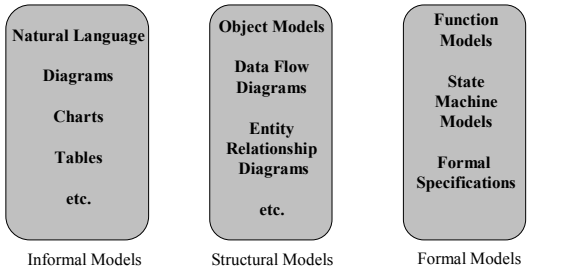
Model Checking and Formality



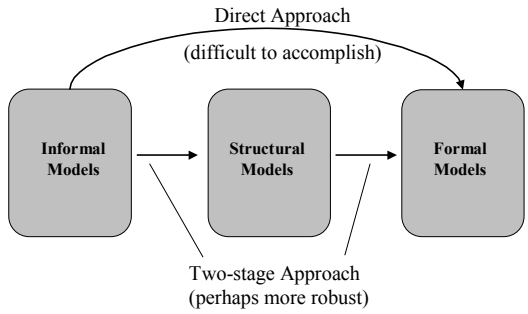
Model Checking and Formality

- Model is more concise and precise than natural language, pseudo-code, and diagrammatic representations
- Model can be used to calculate and predict system behavior
- Model can be analyzed using mathematical reasoning -- proving properties, deriving new behaviors, etc.

Model Checking and Formality

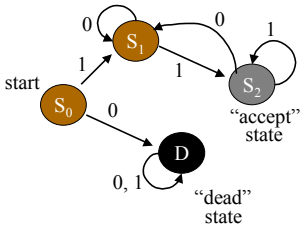


Model Checking and Formality



Model Checking and Formality

A Simple String Parser: Given an input string of 0's and 1's, determine if the string starts and ends with a 1.



State Transition Function

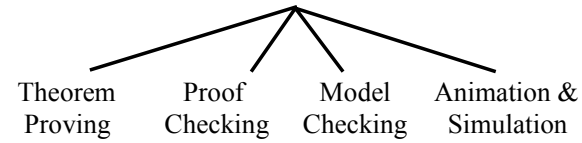
input \ current state	S_0	S_1	S_2	D
0	D	S_1	S_1	D
1	S_1	S_2	S_2	D

next state

Formality

Formal analysis refers to tool-based techniques used to explore, debug, and verify formal specifications.

Methods for Formal Analysis



our focus here

Formality-Properties

- Consistent -- means it is not possible to derive a statement and it's negation within the system
- Complete -- means every true statement within the system is provable
- Decidable -- means there is an effective algorithm (e.g. computer program) for determining whether any statement formed within the system is true
- A system must be consistent to be useable in formal methods (or any other area). While decidability and completeness would be nice, these can not be achieved in most interesting formal systems. However, this does not prevent the effective use of these systems.

Formality

- A sequent is written $\Gamma \vdash \Delta$, which means $\wedge \Gamma$ implies $\vee \Delta$, where Γ is a (possibly empty) list of formulas $\{A_1, \ldots, A_n\}$ and Δ is a (possibly empty) list of formulas $\{B_1, \ldots, B_n\}$
  - the formulas in Γ are called the antecedents
  - the formulas in Δ are called the succedents or consequents
- To restate, $\Gamma \vdash \Delta$ means

  $A_1 \wedge \ldots \wedge A_n$ implies $B_1 \vee \ldots \vee B_n$

Formality

- A sequent calculus proof is a tree of sequents whose root is a sequent of the form $\vdash T$ where T is the formula to be proved and the antecedent is empty
- The proof tree is then generated by applying inference rules of the form:

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

- Intuitively, this rule replaces a leaf node in the proof tree of form $\Gamma \vdash \Delta$ with the *n* new leaves specified in the rule. If *n* is zero, that branch of the proof tree terminates.

Formality

- The Propositional Axiom (*Prop_Axiom*) is one of the rules of inference in the sequent calculus. It has the following form form:

$$\frac{}{(\Gamma, A) \vdash (A, \Delta)}\ \textbf{Prop\_Axiom}$$

- Intuitively, this rule indicates that a proof branch is complete when the sequent above is derived. Note that the consequent means the following:

  $\Gamma \wedge A$ implies $A \vee \Delta$

  which is obviously true.

Formality

- The Rule for Conjunction on the Right (*And_Right*) is another of the rules of inference in the sequent calculus. It has the following form:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash (A \wedge B), \Delta}$$ **And_Right**

- This rule is typical of many sequent calculus inference rules which divide, but simplify, a branch of the proof tree. Note that the consequent is replaced by two simpler formulas which will be easier for a mechanized theorem prover to deal with.

Formality

- The Rule for Conjunction on the Left (*And_Left*) is another of the rules of inference in the sequent calculus. It has the following simple (non-branching) form:

$$\frac{A, B, \Gamma \vdash \Delta}{(A \wedge B, \Gamma) \vdash \Delta}$$ **And_Left**

- This rule is typical of several sequent calculus inference rules which simply restate the "obvious," thereby providing a form easier for a mechanized theorem prover to deal with.

Formality

- The Rule for Implication on the Left (*Implies_Left*) is another of the rules of inference in the sequent calculus. It has the following form:

$$\frac{\Gamma \vdash A, \Delta \qquad B, \Gamma \vdash \Delta}{(A \Rightarrow B, \Gamma) \vdash \Delta}$$ **Implies_Left**

- Similar to the *And_Right* rule, this rule again splits the proof into two cases, each of which will be easier for the mechanical prover to deal with.

Formality

- The Rule for Implication on the Right (*Implies_Right*) is another of the rules of inference in the sequent calculus. It has the following form:

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash (A \Rightarrow B, \Delta)}$$ **Implies_Right**

- This rule does not branch, but provides a form easier for a mechanized theorem prover to deal with.

Formality

- The following example proof in the sequent calculus (taken from NASA Guidebook: *NASA-GB-001-97, Release 1.0, pp. 97-101*) will use only the five sequent calculus inference rules we define earlier -- *Prop_Axiom, And_Left, And_Right, Implies_Left,* and *Implies_Right.*
- The theorem (assumed to be named "Theorem 1") to be proved is the following:

Theorem 1: $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \wedge Q) \Rightarrow R)$

Formality

Theorem 1: $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \wedge Q) \Rightarrow R)$

We begin the proof by forming the requisite sequent:

Antecedents:
none

Consequents:
Formula 1: $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \wedge Q) \Rightarrow R)$

Formality

As our first step we apply the rule *Implies_Right.* This rule will decompose the entire formula. Remember there is an implied "implies" in the sequent. In other words this sequent could be written |- (P => (Q => R)) => ((P ∧ Q) => R). Hence, the implies we apply the rule to is the outside implies on the right of the sequent

Antecedents:
Formula 1: P => (Q => R)

Consequents:
Formula 1: (P ∧ Q) => R

Formality

A second application of the rule *Implies_Right* will decompose the formula below the line in a similar way. Remember that rules applying to the "left" part of the sequent work on formulas above the bar; rules applying to the "right" part of the sequent work below the bar.

Antecedents:
Formula 1: P => (Q => R)
Formula 2: P ∧ Q

Consequents:
Formula 1: R

Formality

We next apply the rule *And_Left* -- this rule will modify (rewrite) Formula 2 above the line. Remember that all formulas above the line are connected by AND's; formulas below the line are connected by OR's.

Antecedents:
Formula 1: P => (Q => R)
Formula 2: P
Formula 3: Q

Consequents:
Formula 1: R

Formality

We next apply the rule *Implies_Left* -- this rule will modify Formula 1 above the line. Remember that *Implies_Left* splits the proof tree into two branches. We show and deal with *Case 1* first, then move to *Case 2* later.

Case 1:
Antecedents:
Formula 1: Q => R
Formula 2: P
Formula 3: Q

Consequents:
Formula 1: R

Formality

To modify Formula 1 above the line, we next apply the rule *Implies_Left* again. Again this splits the proof tree into two branches. We show and deal with *Case 1.1* first, then move to *Case 1.2* later.

Case 1.1:
Antecedents:
Formula 1: R
Formula 2: P
Formula 3: Q

Consequents:
Formula 1: R

Case 1.1 will now yield to an application of *Prop_Axiom*

Formality

As noted, an application of *Prop_Axiom* (Step 5) completes *Case 1.1.* We now move to *Case 1.2*. This is the second case resulting from the application of *Implies_Left* on the *Case 1* sequent. Another application of *Prop_Axiom* (Step 6) completes *Case 1.2* (and in turn *Case 1* itself).

Case 1.2:
Antecedents:
Formula 1: P
Formula 2: Q

Consequents:
Formula 1: Q
Formula 2: R

Case 1.2 will also yield to an application of *Prop_Axiom*

Formality

Having completed the proof for *Case 1*, we now move to *Case 2*. Recall that this is the second case resulting from our first application of *Implies_Left*. Another application of *Prop_Axiom* (Step 7) completes Case 2 (and in turn the entire proof).

Case 2:
Antecedents:
Formula 1: P
Formula 2: Q

Consequents:
Formula 1: P
Formula 2: R

Case 2 will also yield to an application of *Prop_Axiom*

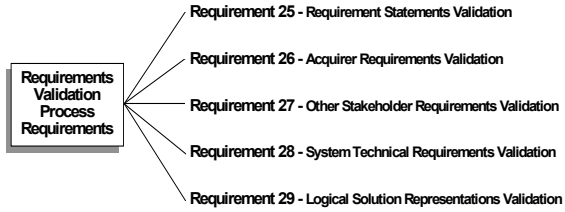
Prototyping

- Oriented to design model
- Dont confuse with simulation
- Some consider functional requirements only.
- Can be a partial implementation of requirements
- Can be an executable specification

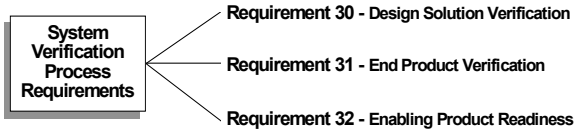
The Two V's and techniques

- V&V in order to avoid what is >verifivation and validation : an eternal debate
- In either case : We check with respect to something.
  - Consider a Petri net model of list automation
  - Verifying Properties of Petri nets does not mean the user is satisfied

V&V in EIA 632



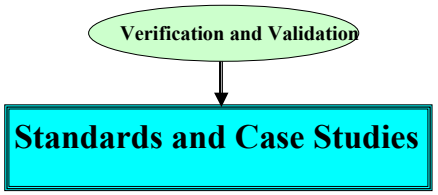
V&V in EIA 632



Conclusions

- Many techniques
- Many tools
- Notation oriented in some cases (formal and semi-formal methods)

Next lecture



•'Validation: Are we building the right product' 'Verification: Are we building the product right'