Overview of Formal Methods

Topics
- Introduction and terminology
- FM and Software Engineering
- Applications of FM
- Propositional and Predicate Logic
- Program derivation
- Intuitive program verification
- Algebraic Specifications
- Overview of Specification languages

Terminology
- **Methods**: general guidelines governing an activity
- **Techniques**: are technical, mechanical, approaches
- **Methodologies**: combine methods, techniques
- **Tools**: can be built to support methodology

Components of a Formal Method
- **Formal systems**: formal languages with well-defined syntax
- **Development technique**: implementation produced from specification
- **Verification technique**: verify implementation satisfies specification

Formal vs. Rigorous
- **Formal**: based on mathematics (including logic), validity of statements can be mechanically checked
- **Rigorous**: strictly follows the rules, compliance can be audited

Important characteristics of FM
- Abstraction
- Proof obligations
- Tool support
- Systematic process
**FM does not replace testing!**

- Reduces burden on testing phases to detect all critical errors
- Facilitates more effective allocation of testing resources
- Can guide the selection of test cases

**Why Use Formal Methods**

- Improve quality of software system
- Fitness for purpose
- Maintainability
- Ease of construction
- Higher confidence in software product
- Reveal ambiguity, incompleteness, and inconsistency in system
- Detect design flaws
- Determine correctness

**V&V and Traceability**

- The Real World
- Validation
- Formal Specification
- Verification
- Code
- Traceability
- Code

**Potential solutions?:**

- Need experimental evidence on large projects.
- Construction of support tools
- Early education???
- Integration of formal methods in more than one phase of software engineering
- Improved (automated) theorem proving strategies
- Handle more than just functional properties
- MOST IMPORTANTLY: do not verify “after the fact”

**Traditional verification techniques not successful.**

Why not?
- Too much like math? (proofs, ugh!)
- Notation too hard to use
- Notation too hard to write out
- “Simple” things take a lot of effort
- “Complex” things seem impossible
- Program verification is an undecidable problem
- “If it works, why mess with it?”
When and Where?

- Introduce FM into existing systems
  - Verify critical properties
  - Facilitate maintenance and reimplementation
- Introduce FM into new systems
  - Capture requirements precisely
  - Reduce ambiguity
  - Guide software development process
  - Basis for testing
  - Formalize requirements analysis and design

Rushby’s “Levels of Rigor”

- Level 0: No use of formal methods.
  - structured walk throughs, ‘formal’ inspections
- Level 1: Use of concepts and notation from discrete mathematics.
  - cleanroom, SCR (software cost reduction)
- Level 2: Use of formalized specification languages with some mechanized support tools.
  - specification languages, ‘rigorous’ proofs
- Level 3: Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking.

Formal Semantics

- Formal semantics provide precise, machine-independent concepts
- Provide unambiguous specification techniques and a rigorous theory to support reliable reasoning.
- A formal definition of a language can suggest a method for constructing programs guaranteed to conform to their specifications.
- So, the purpose of formal specification is ...

Purpose of Formal Specification

- The purpose of a formal specification is to state what a system should do without describing how to do it
- A formal specification may define a system as an abstract datatype.
- A formal specification should avoid implementation bias.

Formal Specifications

- Formal specifications serve as a
  - contract
  - documentation
  - means of communication between client, specifier, and implementer
- Formal specifications are amenable to machine analysis and manipulation

Too Little and Too Much

- There exists a balance between saying enough in a specification and saying too much.
  - say enough so that implementers do not choose unacceptable implementations
  - specifications should capture the requirements completely
  - avoid implementation-bias by not restricting freedom of later designers
### Operational Approach

- Define an abstract machine having states, possibly several components, and some set of primitive instructions.
- Define the machine by specifying how the components of the state are changed by each instruction.
- Define the semantics of a particular programming language in terms of states.
- Abstract machines may be unrealistic from a practical point of view, but the simplistic definition prevents misunderstanding code later.

### Operational Approach con’t

- The semantic description of the programming language specifies a translation into this code.
- Trace through the translated program step-by-step to determine its precise effect.
- Languages defined in this way include PL/I (by the VDM method).

### The Axiomatic Approach

- Associate an "axiom" with each kind of statement in the programming language
  - state what may be asserted after execution of that statement in terms of what was true before
  - an example is the use of pre- and postconditions.

### Another View

- Model-Oriented: define system behavior by constructing model of system in terms of mathematical structures
  - tuples, functions, sets, or sequences
  - languages include VDM, Z, CSP, and Petri Nets
- Property-Oriented: define system behavior indirectly by stating a set of properties that the system must satisfy

### Two Types of Property-Oriented Approaches

- Axiomatic: use first-order predicate logic (pre- and postconditions)
- Algebraic: use axioms in equational form to describe properties

### Obvious Applications

- Computer Security
- Fault-tolerant systems (e.g. Nuclear reactors)
- Safety-critical system (e.g. diagnostic X-ray machine)
- Gain insight into hardware/software systems (e.g. oscilloscope)
- Basically, wherever the cost of failure is high:
  - including systems that are critical in some way
  - replicated many times
  - fixed into hardware, or
  - dependent on quality for commercial reasons
Relevant Areas of Research

- Programming environments
- Formal methods in software development
- Tools that support construction of formal specifications
- Design tools that will generate formal specifications
- Problem/specification decomposition
- Procedural and data abstraction
- Synthesis of efficient code
- "Smart" user interfaces (user-friendly ones!!)
- Methods for determining reuse (of design/specifications/code)