Topics in Formal Methods for Software Development

Dr. Cheng
CSE914
Fall 2002

Acknowledgements

- L. Dillon
- Radu Negulescu
- Anita Jones
- Iliano Cervesato

Topics

- Introduction and terminology
- FM and Software Engineering
- Applications of FM
- Overview of Specification languages
**Terminology**

- **Methods:**
  - general guidelines governing an activity
  - rigorous, systematic, and may be formal
- **Techniques:**
  - are technical, mechanical, approaches
  - may have restricted applicability
- **Methodologies:** combine methods, techniques
- **Tools:** can be built to support methodology

**Components of a Formal Method**

- Formal systems.
  - formal languages with well-defined syntax
  - well-defined semantics
  - proof systems
- Development technique.
  - implementation produced from specification
  - application of development steps
  - refinement process
- Verification technique.
  - verify implementation satisfies specification
  - verify each development step

**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
- Traceability
- Code

---

**Traditional verification techniques not successful.**

Why not?

- Too much like math?
- 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?"
**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"

**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.
  - unstructured 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.

Potential Methods

- Pre- and Post-conditions for specifications
- Vienna Development Method (VDM)
- Rewriting techniques and transformational
- Object-Oriented analysis, design and programming
- Integration of the construction of program and its proof of correctness

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:
  - 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
- Problemspecification decomposition
- Procedural and data abstraction
- Synthesis of efficient code
- "Smart" user interfaces (user-friendly ones!!)
- Methods for determining reuse (of design/specifications/code)

Specifying and verifying security properties

Formal Specifications

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 data type.
- 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

General Approaches

- Operational specifications:
  - Specifies behavior of system through state-based view (e.g., FSA)
- Denotational Specifications:
  - Specifies behavior of system by giving the meaning for various constructs
  - Construct defined in terms of its components
- Axiomatic Specifications:
  - Explains what is true at various points in a system (e.g., pre/post-conditions, invariants)
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)

Denotational Approach

- “Semantic valuation functions” map syntactic constructs in the program into the abstract values that they denote.
- These valuation functions are usually recursively defined:
  - The value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents, and
  - It is this emphasis on the values denoted by all these constructs that gives the approach its name.
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
Intuitive Program Verification

Program Verification and Proofs of Correctness

- Reasoning about the behavior of programs is an old idea. One of the first modern approaches was Floyd’s flowchart annotation scheme.

- Idea: Use logic to make assertions about the functionality of program statements.

- In other words: Formally specify the desired functionality, then verify that the program is a correct implementation of the specification.

Two Questions

- We know that the execution of some programs does not terminate. What specification(s) does a nonterminating program satisfy?

- Verifying programs using only paper and pencil is tedious. How could a Verification Condition Generator (VCG) be built to automatically process code and logical assertions?
Rushby on Formal Verification

- "Formal verification is often equated with 'proof of correctness,' but this simultaneously over- and underestimates the benefits that formal verification can provide. It overestimates the benefits because 'correctness' suggests an absolute guarantee that is impossible for any enterprise that uses modeling to predict the behavior of real-world artifacts."
- J. Rushby, 1993

Partial and Total Correctness

- Partial correctness: If the precondition is true, and the program terminates, then the postcondition is satisfied.
- Total correctness is partial correctness, plus a proof of termination.

Hoare’s Rules

- Program fragments and assertions are composed into triples

\[ \{ P \} S \{ Q \} \]

where
- \( P \) is the precondition assertion,
- \( Q \) is the postcondition assertion, and
- \( S \) is program statements.
Hoare’s Rules

{P} S {Q}

- Interpretation: If P is true before S is executed, then when S terminates, Q is satisfied.
- Triples can be used to define the semantics of programming languages.


Why Use FM When You Know You Aren’t Going To Do The Proofs?

- Syntax and type checking are simple and important.
- The formalized process improves quality.
- Use specifications as the basis for prototypes.

Guidelines

1. Prioritize desired properties and formalize critical correctness requirements at the same level of abstraction as your specification.
2. Record between-level mappings.
3. Know your limitations, that is, know what is not formally defined.
4. Don’t have to prove theorems – symbolic execution or specification-derived test cases are valuable.
Real-world Issues

Andrew Topper states criteria for a method to be formal:
- Documented; if already documented it should be reviewed for improvement
- Taught to new development staff
- Used widely and commonly in the organization
- Measured, both the artifacts or deliverables, and the process itself.


Topper's FM Myths

Myth #1: Using formal methods means following a thick, arduous document with every detail and task spelled out to the nth degree.

Myth #2: Using a formal method means abruptly stopping what we are doing so we can study the process until hell freezes over and we won't get any work done until then.

Myth #3: Using a defined development process means we all have to agree on the "best" method and apply it rigorously without question.

Rushby on Why Use Formal Methods

"The general advantages claimed for all formal methods are that they enable faults to be detected earlier than would otherwise be the case. This is because they allow greater precision and explicitness to be achieved earlier in the lifecycle than otherwise, and because they can support more systematic analyses than informal methods. Stronger forms of this claims are that formal methods cause faults to be detected with greater certainty than would otherwise be the case, and that in certain circumstances, subject to certain caveats, they can guarantee the absence of certain faults. These stronger claims are generally associated with formal methods that undertake "rigorous" or mechanically-checked proofs."

References and Pointers


More References


Even More References