Introduction to Formal Specification

JUNBEOM YOO
jbyoo@knokuk.ac.kr
A Specifier’s Introduction to Formal Methods

Jeanette M. Wing, Carnegie Mellon University

Reference

"A Specifier’s Introduction to Formal Methods"

- Jeannette M. Wing, Carnegie Mellon University
- IEEE COMPUTER, 1990
Contents

• Overview of Formal Methods
• Formal Specification Language
• Pragmatics
• Some Examples
• Bounds of Formal Methods
• Concluding Remarks
Overview of Formal Methods

- Definition
- Features
- Applying Scope
- Pragmatic Considerations
Definition

• **Formal Methods**
  – Mathematically based techniques for describing system properties
    • Have a sound mathematical basis
    • Typically given by a formal specification language

  – Provide frameworks for systematically
    • Specifying,
    • Developing, and
    • Verifying systems
Features

- Formal methods provide means of precisely defining notions like
  - Completeness
  - Consistency
  - Specification
  - Implementation
  - Correctness

- Formal methods address a number of pragmatic considerations
  - Who
  - What
  - When
  - How it is used?
  - ex) System designers use a formal method to specify a system's desired behavioral and structural properties.
Applying Scope

• Any stage of system development can make use of formal methods
  1. Initial statement of a customer’s requirements
  2. System design
  3. Implementation
  4. Testing
  5. Debugging
  6. Maintenance
  7. Verification
  8. Evaluation

• When used early,
  – Can reveal design flaws

• When used later,
  – Can help determine the correctness of a system implementation
  – Can help determine the equivalence of different implementations
Pragmatic Considerations

• **Pragmatic considerations**
  – A set of guidelines
  – Formal methods *should* tell the user
    1. Circumstances under which the method should and can be applied
    2. How it can be applied most effectively

• **Formal Specification**
  – One tangible product of applying formal methods
  – More precise and concise than informal specifications
  – A formal method’s specification language may have **Tool Supports**
    1. Syntax analysis
    2. Semantic analysis with machine aids

Formal Specification:
Use mathematics to *specify* the desired properties of a computer system with support of automatic tools.
Formal Specification Language

- Definition
- Syntactic Domains
- Semantics Domains
- Satisfies Relation
- Properties of Specifications
- Proving Properties of Specificands
Definition

- **Formal specification language:**
  
  \[< \text{Syn}, \text{Sem}, \text{Sat} >, \text{where} \]
  
  - \text{Syn}: syntactic domain
  - \text{Sem}: semantic domain
  - \text{Sat}: \text{Sat} \subseteq \text{Syn} \times \text{Sem}
    
    - \text{syn} is a specification of \text{sem}
    - \text{sem} is a specificand of \text{syn}

- **Considerations**
  - In principle, a formal method is based on some well-defined formal specification language
  - Formal specification language provides a formal method’s [mathematical basis](#)
  - Formal methods differ because their specification languages have different syntactic and/or semantic domains
Syntactic Domains

• **Syn**
  
  – a set of symbols
    
    • Constants
    • Variables
    • Logical connectives
  
  – a set of grammatical rules for combining symbols into well-formed sentences (semantics)
    
    • Ex) $\forall x. P(x) \Rightarrow Q(x)$ : correct!!
    $\forall x. P(x) \Rightarrow Q(x)$ : wrong!!

  – Visual Specification: Graphical elements are also available
    
    • boxes, circles
    • lines, arrows

  – called Specification
Semantic Domains

- **Sem**
  - Formal specification languages differ most in their choice of **semantic domains** (Specificand) such as:
    - Abstract-data-type specification languages
      - algebra, theory, program
    - Concurrent and distributed systems specification languages
      - state sequence, event sequence, state and transition sequence
      - stream, synchronization tree, partial order
      - state machine
    - Programming languages
      - function from input to output, computation
      - predicate transformation
      - relation, machine instruction
      - called **Implementation**
Satisfies Relation

- \textit{Sat}:
  - Specifies different aspects of a single specificand using different specification languages:
    1. \textbf{Behavioral specification aspect}
       - Constraints on observable behavior of specificands
       - System’s required functionality (mapping from inputs to outputs)
       - Others: fault tolerance, safety, security, response time, space efficiency
    2. \textbf{Structural specification aspect}
       - Constraints on the internal composition of specificands
       - Various hierarchical and uses relations
       - Call graph, data-dependency diagram, definition-use chain
Properties of Specifications

• Specification language should be defined as
  1. Unambiguous
     • If and only if it has exactly one meaning
     • Any natural languages and graphs are not formal inherently
  2. Consistent
     • If and only if its specificand set is non-empty
     • Cannot derive anything contradictory from the specification
     • There is some implementation that will satisfy the specification
  3. Complete
     • Need not be complete in the sense used in mathematical logic
     • Relatively-completeness properties might be desirable
     • In practice, we must usually deal with incomplete specifications

• A specification has implementation bias if it places unnecessary constraints on its specificand
Proving Properties of Specifications

- Most formal specification languages have logical inference systems
  - Can prove properties from the specification about specificands
  - Can predict system’s behavior without executing or building the system
  - Can be mechanized
    - Theorem proving
    - Model checking
    - called Formal Verification (Part II)
Pragmatics

Users
Uses
Characteristics
Users

• 5 kind of users
  1. Specifier : write, evaluate, analyze, and refine specifications
  2. Customer : hired the specifiers
  3. Implementer : realize a specification
  4. Client : use a specified system
  5. Verifier : prove the correctness of implementations

• A formal method’s guidelines should identify
  1. Different types of users the method is targeted for
  2. Capabilities the users should have
  3. Application domain of the method
Uses

• The greatest benefit comes
  – from the process of formalizing
  – rather than the end result

• Can apply formal methods in all phases of SW development
  1. Requirements analysis
  2. System design
  3. System verification
  4. System validation
  5. System documentation
  6. System analysis and evaluation

• These applications should be considered as an integral one, framework
Uses 1. Requirements Analysis

- Formal methods help clarify customer’s informally stated requirements
  - Crystallize customer’s vague ideas
  - Reveal
    - Contradictions,
    - Ambiguities, and
    - Incompleteness in the requirements

- On the specification, both customers and specifiers can see
  - Whether it reflects customer’s intuition
  - Whether specific and set has desired set of properties
Uses 2. System Design

• **Two important activities** during design
  1. Decomposition
  2. Refinement

• **Decomposition**
  – Process of partitioning a system into smaller modules
  – Interface specifications specify interfaces between modules

• **Refinement**
  – Process of refining modules at one level to modules at a lower level
  – Each refinement step should prove that a specification (program) at one level satisfies a higher level specifications
    • Program transformation, Program synthesis, Inferential programming
  – Formal methods and formal specification languages can state **proof obligations (assumptions)** precisely
Uses 3. System Verification

- System verification
  - Showing that a system satisfies its specification

- **Formal Verification**
  - Using formal specifications to verify a system
  - Cannot completely verify an entire system,
  - But can certainly verify smaller and critical part of system.
    - Gypsy, HDM(Hierarchical Development Method), FDM(Formal Development Method)
    - M-EVES(Environment for Verifying and Emulating Software)
    - HOL(Higher Order Logic)

- Difficulties in formal system verification
  - Should state explicitly assumptions about its environment : Not easy!
    - “Bounds of Formal Methods”
Uses

4. System Validation

- Formal methods can aid in system testing and debugging.

- Specification alone:
  - Used to generate test cases for black-box testing
  - For boundary condition tests

- Specification along with implementation:
  - Used to generate test cases
  - Additionally, can be used for testing analysis
    - Path testing
    - Unit testing
    - Integration testing
    - Etc.
Uses 5. System Documentation

• Formal specification
  – Captures “What” rather than “How”
  – Serves as a communication medium between
    • Clients and Specifiers
    • Specifiers and Implementers
    • Among members of an implementation team

- System analysis and evaluation
  - After system has been built and tested,
  - Critical analysis of its functionality and performance should be done
    - Does the system do what the customer wants?
    - Does it do it fast enough?
  - Formal method used in the development can help formulate and answer these questions

- Most formal methods have not yet been applied to specifying large-scale software and hardware systems
  - Size of the specification
  - Complexity of the specificand
    - Internal complexity
    - Interface complexity
Characteristics

• Formal method’s characteristics influence the style in which a user applies it
  – Whether its language is graphical or textual
  – Whether its underlying logic is first-order or high-order
  – Etc.

• Formal method reflects a combination of many different characteristics:
  1. Model-oriented vs. Property-oriented
  2. Visual languages
  3. Executable
  4. Tool-supported
Characteristics 1. Model-oriented vs. Property-oriented

• **Model-oriented methods**
  – Define system’s behavior **directly** by constructing a model of the system
  1. For sequential systems
     • Parnas’ state machines, VDM, Z, SCR, NuSCR
  2. For concurrent and distributed systems
     • Petri Nets, CCS, Hoare’s CSP, Unity, I/O automata
     • Temporal logic, Lamport’s transition axiom method, LOTOS

• **Property-oriented methods**
  – Define system’s behavior **indirectly** by stating a set of properties using axioms
  1. Axiomatic methods
     • Iota, OBJ, Anna, Larch
  2. Algebraic methods
     • Act One

---

**Algebraic specification of abstract data types can handle:**
- Error values
- Nondeterminism
- Parameterization
Characteristics: Visual Languages

- Visual specification languages
  - Any one who contains graphical elements in their syntactic domains

- Many examples
  - Petri nets: for concurrent systems
  - Statecharts: for specifying state transitions in reactive systems

- Semiformal methods
  - Multiple interpretations or text attached
  - Jackson’s method (UML)
  - SASD, OOD
  - Requirements Engineering Methodology

Figure 3. State chart specification of a one-slot buffer.
Characteristics 3. Executable

- Executable Specification
  - Can run on a computer

- Specifiers can use executable specifications
  - To gain immediate feedback about the specification itself.
  - To do rapid prototyping
  - To test a specific and through symbolic execution of the specification

- Many examples
  - Statecharts
  - OBJ
  - Prolog, Paisley
  - Most recent ones
Characteristics

4. Tool-supported

• Model-Checking tools
  – Let users construct a finite-state model of the system
  – Then show a property holds in each state or state transition of the system
  – EMC, SMV, SPIN

• Proof-checking tools
  – Let users treat algebraic specifications as rewrite rules
    • Larch Prover, Affirm, Reve
  – Handling first-order logic
    • Boyer-Moore Theorem Prover, FDM, HDM, m-EVES
  – Handling higher-order logic
    • HOL, LCF, OBJ
Some Examples

Abstract Data Type: Z, VDM, Larch
Concurrency: Temporal Logic, CSP, Transition Axioms
Some Examples

• 6 well-known formal methods (in 1990s)
  – Abstract data type: Z, VDM, Larch
    • Symbol table example
  – Concurrency: Temporal Logic, CSP, Transition Axioms
    • Unbounded buffer example

• When specifying the same problem with different methods, they look
  – Remarkably similar
  – Or totally different
  – Due to
    • Nature of the specificand
    • Simplicity of the specificand
    • Methods themselves
Abstract Data Type: Z, VDM, Larch

- 3 different specifications for a symbol table
## Abstract Data Type: Z, VDM, Larch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base</strong></td>
<td>Model-oriented (Also property-oriented)</td>
<td>Model-oriented</td>
<td>Property-oriented</td>
</tr>
<tr>
<td><strong>Readability</strong></td>
<td>Good</td>
<td>Normal</td>
<td>Bad</td>
</tr>
<tr>
<td><strong>Specifiability</strong></td>
<td>Bad</td>
<td>Normal</td>
<td>Good</td>
</tr>
<tr>
<td><strong>Size</strong></td>
<td>Normal</td>
<td>Compact</td>
<td>Long</td>
</tr>
<tr>
<td><strong>Tool-Support</strong></td>
<td>Proof Checker B</td>
<td>N/A</td>
<td>Syntax Analyzer Larch Prover</td>
</tr>
</tbody>
</table>
Concurrency: Temporal Logic, CSP, Transition Axioms

- 3 different specifications for an **unbounded buffer**

---

**Temporal Logic**

- Temporal logic specification of an unbounded buffer.

1. \( (\text{n} \geq m) \Rightarrow (\text{l} \geq l) \)
2. \( (\text{n} \geq m) \land \exists m \in \text{n} \exists! \text{l} \geq l \) \( (\text{l} \geq l) \Rightarrow (\text{n} \geq m) \)
3. \( (\text{l} \geq l) \Rightarrow (\text{n} \geq m) \)
4. \( (\text{l} \geq l) \Rightarrow (\text{n} \geq m) \)

---

**CSP**

- CSP program and specification of an unbounded buffer.

```
BUFFER = P
where P = \text{left} \in \text{int} \rightarrow P_{\text{left}}
and P_{\text{left}} = (\text{left} \in \text{int} \rightarrow P_{\text{left}} \rightarrow \text{right} \in \text{int} \rightarrow P_{\text{right}})

BUFFER sat (\text{right} \leq \text{left}) \land (\text{if right = left then left = ref else right = ref})
```

---

**Transition Axioms**

- Transition axiom specification of an unbounded buffer.

```
module BUFFER with subroutines PUT, GET

state functions:
- buffer : sequence of message
- par : message or NULL
- gval : message or NULL

initial conditions:
- buffer = 0

safety properties
1. \( \lnot \text{after PUT) \Rightarrow \text{par} = \text{PUT\_PAR} \)
   - (after PUT) \Rightarrow \text{par} = \text{NULL}
2. \( \lnot \text{after GET) \Rightarrow \text{gval} = \text{NULL} \)
   - (after GET) \Rightarrow \text{GET\_PAR} = \text{gval}
3. allowed changes to buffer
   - par when \( \text{in} (\text{PUT}) \)
   - gval when \( \text{in} (\text{GET}) \)
   - (BUFFER) \text{PUT} \Rightarrow \text{par} \neq \text{NULL} \Rightarrow \text{par} = \text{NULL} \land \text{buffer} = \text{buffer} + \text{par}
   - (BUFFER) \text{GET} \Rightarrow \text{gval} = \text{NULL} \land \text{buffer} < \text{buffer} = \text{gval}

liveness properties
4. \( \text{PUT} \land \text{buffer} < \text{min} \Rightarrow \text{after PUT} \)
5. \( \text{GET} \land \text{buffer} > 0 \Rightarrow \text{after GET} \)
```
Concurrency: Temporal Logic, CSP, Transition Axioms

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Property-oriented</td>
<td>Model-oriented (for specifying)</td>
<td>Model-oriented (for specifying)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Property-oriented (for proving)</td>
<td>Property-oriented (for proving)</td>
</tr>
<tr>
<td>Readability</td>
<td>Normal</td>
<td>Normal</td>
<td>Good</td>
</tr>
<tr>
<td>Specificability</td>
<td>Bad</td>
<td>Bad</td>
<td>Good</td>
</tr>
<tr>
<td>Size</td>
<td>Compact</td>
<td>Compact</td>
<td>Long</td>
</tr>
<tr>
<td>Tool-Support</td>
<td>Many related tools</td>
<td>Proof Checker B</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Konkuk University
Bounds of Formal Methods

Between the Ideal and Real Worlds
Assumptions about the Environment
Between the Ideal and Real Worlds

- Formal methods are
  - Based on mathematics
  - But not entirely mathematical

- Two important boundaries between the mathematical and the real world

Figure 10. Mapping informal requirements for a formal specification.

Figure 11. Mapping the real world to an abstract model.
Assumptions about the Environment

• There is a boundary between a real system and its environment
  – Environment is out of the scope of formal specifications (Open System)
  – Except, Gist specification language
    • Environment $\Rightarrow$ System
    • Environment is a set of assumptions
    • System is a set of constraints on its behaviors placed by specifiers
  – Implicit assumptions in programming language areas
  – Specifiers should make explicit as many assumptions as possible.

• Hazard Analysis
  – Identify a system’s safety-critical components
    • FTA, FMEA, HAZOP
  – A complementary technique to formal methods
Concluding Remarks

Formal Methods
Challenges
Formal Methods

• In a strict mathematical sense,
  – Formal methods differ greatly from one another
• In a practical sense,
  – Formal methods do not differ radically from one another

• Formal methods can be used
  1. Identify
     • Deficiencies in informal requirements
     • Discrepancies between a specification and an implementation
     • Errors in existing programs and systems
  2. Specify
     • Medium-sized and nontrivial problems
     • Functional behavior
  3. Provide
     • Deeper understanding of the behavior of systems
Challenges

1. Specifying nonfunctional behavior
   - Reliability, safety, real-time, performance, human factors
2. Combining different methods
   - Domain specific + General
   - Formal + Informal
3. Building more usable and robust tools
   - Can manage large specifications
   - Can perform more complicated semantic analysis
4. Building specification libraries
   - Reuse in general or domain-specific purpose
5. Formal methods based software development
6. Scale up existing techniques
7. Educating and training