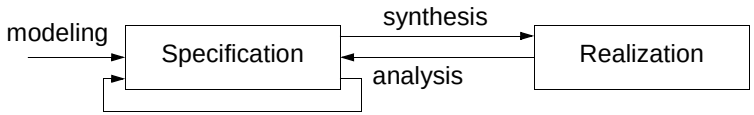
Formal Specification

- Requirements specification
  - notational statement of system services
- Software specification
  - formal abstract depiction of system services
- Architectural specification
  - graphical representation of system structure
  - formal abstract depiction of key system properties
- Module specification
  - formal module interface, behavior, interaction specifications
- Two different concepts
  - >> Formal Methods
  - >> Formal Specification Languages

What Are Formal Methods? (1)

- Formal Method (FM) =
  specification language + formal reasoning
- Body of techniques supported by
  - precise mathematics
  - powerful analysis tools
- Rigorous, effective mechanisms for system
  - modeling
  - synthesis
  - analysis



What Are Formal Methods? (2)

- Use of formalisms
  - e.g., logic, finite state machines, discrete mathematics
- in system descriptions
  - e.g., system models, constraints, specifications, designs
- for a broad range of effects
  - e.g., highly reliable, safe, secure systems
  - e.g., more effective production
- and varying levels of use
  - *guidance*: structuring what to say
  - *documentation*: unambiguous communication
  - *rigor*: formal specification and proofs
  - *mechanisms*: proof assistance, testing

Objectives of Formal Methods (1)

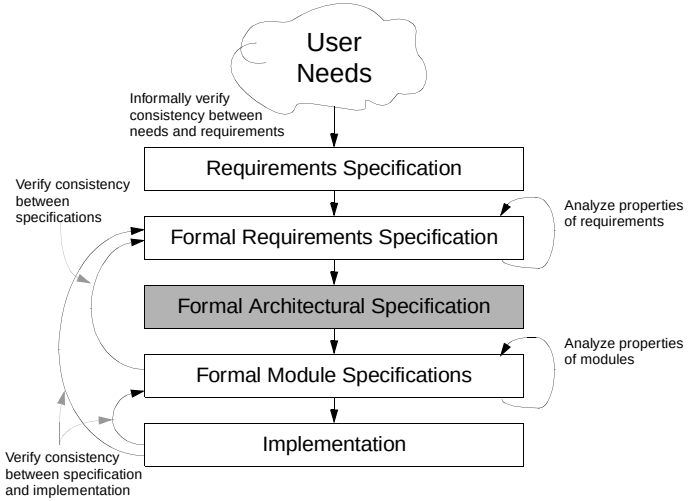
- Requirements specification
  - clarify customer's requirements
  - reveal ambiguity, inconsistency, incompleteness
- System/Software design
  - decomposition
    - structural specifications of component relations
    - behavioral specification of components
  - refinement
    - demonstrating that next level of abstraction satisfies higher level

  >> includes architecture-level design

Objectives of Formal Methods (2)

- Verification
  - "are we building the system right?"
  - proving that a specificand (realization) satisfies its specification
- Validation
  - "are we building the right system?"
  - testing and debugging
  - e.g., use specification to determine test cases
- Documentation
  - communication among stakeholders

Using Formal Methods in Software Development



Why Use Formal Methods? (1)

- Formal methods have the potential to improve both *quality* and *productivity* in software development
  - to circumvent expensive problems in traditional development practices
  - to promote insight and understanding
  - to enhance early error detection
  - to develop safe, reliable, secure software-intensive systems
  - to enable formal modeling and analysis
  - to facilitate verifiability of implementation
  - to enable
    - simulation, animation, proof, execution, transformation*
  - ➔ to maintain competitive advantage by more effective software development process

* depending on method used

Why Use Formal Methods (2)

- Formal methods are on the verge of becoming best practice and/or required practice for developing safety-critical and mission-critical software systems
- To avoid legal liability repercussions
  - reduce risks associated with software development
  - increase safety, security, reliability
- To ensure that systems meet regulations and standards
  - increasing concern with safety by government
  - stay in front of legal and regulatory agencies

Why Not?

- Emerging technology with unclear payoff
- Lack of experience and evidence of success
- Lack of automated support
- Existing tools are user unfriendly
- Ignorance of advances
- High learning curve
- Perfection and mathematical sophistication required
- Techniques not widely applicable
- Techniques not scalable
- Too many in-place techniques and tools

Myths of Formal Methods

- Formal methods can guarantee that software is perfect
  - >> how do you make sure the spec you build is perfect?
- Formal methods are all about program proving
  - >> they are about modeling, communicating, demonstrating
- Formal methods are only useful for safety-critical systems
  - >> may be useful in *any* system (e.g., highly reusable modules)
- Formal methods require highly trained mathematicians
  - >> many methods involve no more than set theory and logic
- Formal methods increase the cost of development
  - >> the opposite is often the case
- Formal methods are unacceptable to users
  - >> users will find them very helpful if properly presented
- Formal methods are not used on real, large-scale software
  - >> they are used daily in many branches of industry

Integrating Formal Methods into Development

- Option 1: business as usual with after-the-fact verification
  - formal specification constructed after system implementation
  - implementation checked for consistency against the spec
  - increases confidence in the system
  - time- and money-consuming
- Option 2: verification in parallel
  - two teams — development team and formal verification team
  - requires constant communication between the two teams
  - may degenerate into option 1 due to poor communication
  - less time consuming but as expensive as option 1
- Option 3: integrated verification
  - one team that does development and formal verification
  - single integrated development process
  - better, cheaper, *and* faster than options 1 and 2

Formal Specifications

- Intended to remedy *the seven sins of the specifier*
  - >> noise / redundancy / remorse
  - >> silence
  - >> overspecification
  - >> contradiction
  - >> ambiguity
  - >> forward reference
  - >> wishful thinking

Desirable Properties of Formal Specifications

- Unambiguous
  - exactly one specificand (set) satisfies it
  - e.g., “Component X has a single port on its top and bottom”
- Consistency
  - a specificand exists that satisfies it
  - e.g., interfaces of interacting components must match
- Completeness
  - all aspects of specificands are specified
  - e.g., interfaces of all components must be specified
  - may be achieved incrementally
- Inference
  - consequence relation used to prove properties about the specificands that satisfy a specification

Formal Specification in Software Development

- Formal specifications ground the software development process in the well-defined basis of computer science
- Orientation goes from customer to developer
- Formal specifications are expressed in languages with formally defined syntax and semantics
  - hierarchical decomposition
  - mathematical foundation
  - graphical presentation
  - accompanied by informal description

Formal Specification Languages

- A formal specification language consists of
  - ***syn***tax — the notation
  - ***sem***antics — the specifiable objects
  - ***sat***isfies — relation defining which objects satisfy which notations
- A formal specification defines
  - syntax — signature of the mapping
  - semantics — meaning of the mapping
  - exceptions — undefined/erroneous mappings
- If *sat(syn,sem)* then
  - syn is a *specification* of sem
  - sem is a *specificand* of syn

Characteristics of Specification Languages

- Model-oriented specifications
  - specify system behavior by constructing a model in terms of well-defined mathematical constructs
- Property-oriented specifications
  - specify system behavior in terms of properties that must be satisfied
- Visual specifications
  - specify system structure and behavior by graphical depictions
- Executable specifications
  - specify system behavior completely enough that specifications can run on a computer
  - >> this is *not* programming

Tool Support for Specification Languages

- Modeling
  - editors
  - word processors
  - editor / word processor plug-ins
- Analysis
  - syntactic checking
  - model checking
  - proof checking
- Synthesis
  - refinement
  - code generation
  - test case generation
  - test oracle generation

Types of Formal Specifications

- Behavioral specifications describe constraints on the behavior of a specificand
  - functionality
  - safety & security
  - performance
- Structural specifications describe constraints on the internal composition of a specificand
  - module interconnection
  - uses and is-composed-of
  - dependence relations
- Interaction specifications describe constraints on the interactions between two or more specificands
  - interface matching
  - protocol matching

Basic Specification Language Types

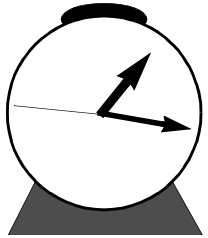
- Axiomatic specifications
  - defines operations by logical assertions
- State transition specifications
  - defines operations in terms of states and transitions
- Abstract model specifications
  - defines operations in terms of a well-defined math model
- Algebraic specifications
  - defines operations by collections of equivalence relations
- Temporal logic specifications
  - defines operations in terms of order of execution and timing
- Concurrent specifications
  - defines operations in terms of simultaneously occuring events

Axiomatic Specifications

- Implicitly defines behavior
  - >> in terms of (first-order) logic formulas
  - >> specifying input/output assertions
  - >> and possibly intermediate assertions
- Specification includes
  - operation interfaces with input/output parameters
  - operation axioms with pre/post assertions on input/output
- Pros and cons
  - fairly easy to understand
  - widely applicable
  - hard to scale up
  - widely used technique in proofs (inductive assertion method)
  - foundation of mathematics in software development
- Languages: VDM, Anna, Z

Example Problem: Clock

- Initially, the time is midnight, the bell is off, and the alarm is disabled
- Whenever the current time is the same as the alarm time and the alarm is enabled, the bell starts ringing
  - >> this is the only condition under which the bell begins to ring
- The alarm time can be set at any time
- Only when the alarm is enabled can it be disabled
- If the alarm is disabled while the bell is ringing, the bell stops ringing
- Resetting the clock and enabling or disabling the alarm are considered to be done instantaneously



Axiomatic Specifications — VDM Clock

```
INIT()
   ext wr time:N, bell:{quiet, ringing},
         alarm:{disabled, enabled}
   pre true
   post (time' = midnight) /\ (bell' = quiet) /\
        (alarm' = disabled)

TICK()
   ext wr time:N, bell:{quiet, ringing}
       rd alarm_time:N, alarm:{disabled, enabled}
   pre true
   post (time' = succ(time)) /\
        (if (alarm_time' = time') /\ (alarm' = enabled)
        then (bell' = ringing) else (bell' = bell))
```


Abstract Model Specifications

- Explicitly describes behavior in terms of a model using well-defined types (sets, sequences, relations, functions) and defines operations by showing effects on model
- Specification includes
  - type — syntax of object being specified
  - model — underlying structure
  - invariant — properties of modeled object
  - pre/post conditions — semantics of operations
- Pros and cons
  - state is made explicit in model
  - suggests an implementation
  - widely applicable because of modeling orientation
- Notations: VDM, Z, RAISE

Abstract Model Specifications — Z Clock

```
BellStatus : {quiet,ringing}
AlarmStatus : {disabled,enabled}
```

```
Clock
time, alarm_time : N
bell : BellStatus
alarm : AlarmStatus
```

```
InitClock
ΔClock
(time' = midnight) /\ (bell' = quiet) /\
(alarm' = disabled)
```

Algebraic Specifications

- Implicitly defines behavior by set of equivalence relations describing properties possessed by the objects and their operations
- Specification includes
  - functionality — syntax and legal constructions
  - relations — semantics by equivalence classes
- Pros and cons
  - only pure functions described (no side effects)
  - supports extensibility of data abstractions
  - often hard to comprehend and construct
  - particularly applicable to ADTs
- Notations: OBJ, Larch, Clear, Anna

Algebraic Specifications — Clock

functionality

```
init: -> CLOCK
tick, enable, disable: CLOCK -> CLOCK
setalarm: CLOCK x TIME -> CLOCK
time, alarm_time: CLOCK -> TIME
bell: CLOCK -> {ringing, quiet}
alarm: CLOCK -> {on, off}
```

relations

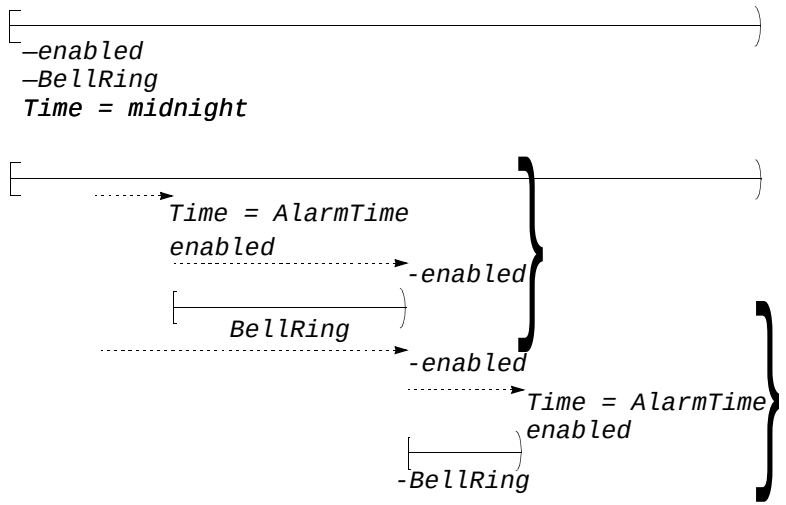
```
time(init) -> midnight
time(tick(C)) -> time(C) + 1
time(setalarm(C,T)) -> time(C)

alarm_time(init) -> midnight
alarm_time(tick(C)) -> alarm_time(C)
alarm_time(setalarm(C,T)) -> T
```


Concurrent and Temporal Specifications

- Explicitly define behavior by descriptions of system states and sets of (timed and/or ordered) events with guards that cause (simultaneous) state transitions
- Specification includes
  - states — possible values
  - transitions — value changes
  - events — causes of transitions
  - ordering and timing — constraints on transitions
- Pros and cons
  - powerful specification mechanism
  - applicable to a large class of existing systems
  - often hard to comprehend and construct
  - spec-level timing is hard to ensure in the implementation
- Notations: CSP, GIL, Petri nets, statecharts, posets

Concurrent & Temporal Specs — GIL Clock



State Transition Specifications

- Explicitly describes system behavior by a set of states and defines operations as transitions between states or observations on state
- Specification includes
  - states — possible values
  - transitions — semantics by state transformations and observations
- Pros and cons
  - free of representational details
  - state explosion is common
  - extensions to minimize states and modularize
  - particularly applicable to control systems and hardware
- Textual and graphical notations
  - StateCharts, ASLAN, Paisley, InaJo, Special

State Transition Specifications — Clock