

# VU LEARNING HUB

[vulearninghub.xyz](https://vulearninghub.xyz)

**HIGHLIGHTED HANDOUT**

## CS709

Formal Methods for Software Engineering

**CS709 by VU Learning HUB**

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

## CS709 - Formal Methods for Software Engineering

A structured study handout compiled from the course's lecture topics. Read each topic's objectives first, study the key concepts, then use the summary and exam tips for revision. ☰

# CS709

## Formal Methods for Software Engineering

Based on 45 lecture topics

Compiled for VU students as a structured study guide

*Concepts are summarized in the writer's own words;  
use alongside your lecture videos for best results.*

# Contents

|                                                               |           |
|---------------------------------------------------------------|-----------|
| <b>Unit 1: Mathematical Foundations</b>                       | <b>3</b>  |
| Topic 1: Why Formal Methods . . . . .                         | 3         |
| Topic 2: Propositional Logic . . . . .                        | 4         |
| Topic 3: Predicate Logic . . . . .                            | 5         |
| Topic 4: Sets, Relations, and Functions . . . . .             | 5         |
| Topic 5: Induction and Proof Techniques . . . . .             | 6         |
| <b>Unit 2: Formal Specification</b>                           | <b>7</b>  |
| Topic 6: What Is a Formal Specification . . . . .             | 7         |
| Topic 7: Z Notation Basics . . . . .                          | 8         |
| Topic 8: VDM and Model-Based Specification . . . . .          | 8         |
| Topic 9: Algebraic Specification . . . . .                    | 9         |
| Topic 10: Specifying Concurrent Systems . . . . .             | 10        |
| <b>Unit 3: Program Verification</b>                           | <b>11</b> |
| Topic 11: Hoare Logic . . . . .                               | 11        |
| Topic 12: Preconditions, Postconditions, Invariants . . . . . | 11        |
| Topic 13: Model Checking . . . . .                            | 12        |
| Topic 14: Theorem Proving . . . . .                           | 13        |
| Topic 15: Refinement: From Spec to Code . . . . .             | 14        |
| <b>Unit 4: Applications and Practice</b>                      | <b>15</b> |
| Topic 16: Verifying Real Software: Case Studies . . . . .     | 15        |
| Topic 17: Alloy and Lightweight Modeling . . . . .            | 15        |
| Topic 18: Tools Tour: What Exists . . . . .                   | 16        |
| Topic 19: Formal Methods in Agile and DevOps . . . . .        | 17        |
| Topic 20: Limits, Criticisms, and the Future . . . . .        | 18        |
| <b>Glossary of Key Terms</b>                                  | <b>18</b> |

# Course Introduction

Formal Methods apply mathematics to software engineering: using logic and precise models to specify what software should do and to prove that it does it. Where testing can only show the presence of bugs, formal verification can prove their absence - crucial for safety-critical systems like aircraft controls, medical devices, and cryptographic protocols.

The course begins with the mathematical foundations: propositional and predicate logic, sets, relations, and functions - the language in which specifications are written. Students then learn formal specification techniques, from model-based approaches (Z notation, VDM) to algebraic and process-algebraic methods.

Later units cover program verification: Hoare logic, preconditions and postconditions, loop invariants, and model checking - the automated verification technique behind many industrial successes. The course closes with practical questions: when formal methods pay off, what tools exist, and how they integrate with modern development.

The emphasis throughout is on precision as an engineering tool: mathematics that finds bugs before they fly.

**How to use this handout:** Each topic lists its learning objectives, explains the key concepts in simple words, defines important terms, gives an example where helpful, and closes with a summary plus exam tips. Topics marked **Very important** deserve extra revision.

## Unit 1: Mathematical Foundations

### Topic 1: Why Formal Methods

#### Learning Objectives

- Define formal methods
- Explain their value proposition
- Identify suitable application domains

#### Key Concepts

Formal methods are mathematically-based techniques for specifying, developing, and verifying software. They replace ambiguous natural-language requirements with precise models that can be analyzed - even proved correct.

The value: testing samples behaviors, but formal verification covers all behaviors within the model. For safety-critical systems (avionics, railway signaling, medical implants), the cost of failure justifies the cost of proof. Famous successes include verified microkernels (seL4), Paris Metro signaling, and cryptographic protocols.

Limitations are real: formal methods cost expertise and time, models may mismatch reality, and they suit critical components better than whole CRUD apps. The engineering judgment is where to apply them.

#### Key Points

- Formal methods: mathematics for specifying and verifying software.
- Testing finds bugs; verification proves absence (within the model).

- Best for safety/security-critical components; costly elsewhere.

### Key Terms

**Formal method:** A mathematically-based specification/verification technique.

**Verification:** Proving a system meets its specification.

**Safety-critical:** Systems whose failure endangers life.

### Example

- *The seL4 microkernel's proof guarantees its security properties mathematically.*

**Summary:** Formal methods trade effort for certainty where failure is unacceptable.

### Exam Tips

- ★ Very important: testing vs. verification distinction.
- ★ Remember suitable domains.

## Topic 2: Propositional Logic

### Learning Objectives

- Explain propositions and connectives
- Use truth tables
- Identify tautologies and contradictions

### Key Concepts

Propositional logic reasons about statements (propositions) that are true or false. Connectives build compound statements: negation (not  $p$ ), conjunction ( $p$  and  $q$ ), disjunction ( $p$  or  $q$ ), implication ( $p$  implies  $q$ ), and equivalence ( $p$  iff  $q$ ).

Truth tables enumerate all cases: implication is false only when  $p$  is true and  $q$  false - the source of many beginner confusions ('if false then anything' is vacuously true). Tautologies (always true, e.g.  $p$  or not  $p$ ) and contradictions (always false) are identified this way.

Logical equivalences (De Morgan's laws, distributivity) let us transform specifications - the algebra underlying circuit design and program reasoning. Satisfiability (is there an assignment making it true?) drives SAT solvers, modern verification engines.

### Key Points

- Propositions with connectives: not, and, or, implies, iff.
- Truth tables decide tautology, contradiction, satisfiability.
- Equivalences (De Morgan) transform formulas algebraically.

### Key Terms

**Proposition:** A statement that is true or false.

**Tautology:** A formula true under all assignments.

**Satisfiability:** Existence of a truth assignment making a formula true.

### Example

- *not ( $p$  and  $q$ ) is equivalent to (not  $p$ ) or (not  $q$ ) - De Morgan.*

**Summary:** Propositional logic is the calculus of true/false reasoning.

## Exam Tips

- ★ Very important: implication's truth table.
- ★ Remember De Morgan's laws.

## Topic 3: Predicate Logic

### Learning Objectives

- Explain predicates and quantifiers
- Translate statements to predicate logic
- Discuss limitations

### Key Concepts

Predicate logic adds predicates (properties like  $\text{Prime}(x)$ ), variables, and quantifiers: universal (for all  $x$ ) and existential (there exists  $x$ ). 'All humans are mortal' becomes: for all  $x$ ,  $\text{Human}(x)$  implies  $\text{Mortal}(x)$ .

Quantifier order matters: 'for all  $x$  there exists  $y$ ' differs from 'there exists  $y$  for all  $x$ ' - the classic scope distinction. Nested quantifiers express complex specifications precisely.

Predicate logic can express most mathematics, but full predicate logic is undecidable - no algorithm decides all truths. Practical verification uses decidable fragments (propositional logic, certain theories) plus interactive theorem provers for the rest.

### Key Points

- Predicates + quantifiers (forall, exists) express rich statements.
- Quantifier order changes meaning.
- Full predicate logic is undecidable; tools use fragments.

### Key Terms

**Predicate:** A property or relation applied to terms.

**Universal quantifier:** For all: asserts about every value.

**Existential quantifier:** There exists: asserts some value.

### Example

- *forall  $x$ .  $\text{Human}(x) \rightarrow \text{Mortal}(x)$*

**Summary:** Predicate logic precisely captures 'for all' and 'there exists' reasoning.

### Exam Tips

- ★ Very important: quantifier order with an example.
- ★ Remember undecidability.

## Topic 4: Sets, Relations, and Functions

### Learning Objectives

- Use set notation and operations
- Define relations and their properties
- Explain functions formally

### Key Concepts

Sets are collections: {1, 2, 3}, with operations union, intersection, difference, and power set. Specifications model system states as sets and operations as set transformations.

Relations are subsets of Cartesian products:  $R \subseteq A \times B$ . Properties - reflexive, symmetric, transitive - classify relations; equivalence relations partition sets (crucial for state minimization); partial orders model hierarchies.

Functions are relations assigning each input exactly one output. Injective (one-to-one), surjective (onto), bijective functions; composition; inverse functions. Program semantics treats programs as functions from states to states - the mathematical view of computation.

### Key Points

- Sets with union/intersection/difference model system states.
- Relation properties: reflexive, symmetric, transitive.
- Functions: injective, surjective, bijective; programs as state functions.

### Key Terms

**Relation:** A subset of a Cartesian product.

**Equivalence relation:** Reflexive, symmetric, transitive relation.

**Bijection:** A one-to-one onto function.

### Example

- 'Same age as' is an equivalence relation on people.

**Summary:** Sets, relations, and functions are the vocabulary of formal specification.

### Exam Tips

- ★ Key point: relation properties with examples.
- ★ Remember function types.

## Topic 5: Induction and Proof Techniques

### Learning Objectives

- Prove by mathematical induction
- Explain structural induction
- Outline proof strategies

### Key Concepts

Mathematical induction proves  $P(n)$  for all naturals: show  $P(0)$  (base), then  $P(k)$  implies  $P(k+1)$  (step). It mirrors recursion - the natural proof method for recursive programs and data structures.

Structural induction generalizes to recursively-defined structures (lists, trees): prove for base constructors, then for composite ones assuming the parts. This proves properties of algorithms on trees and expressions.

Proof strategies: direct proof, contrapositive (prove not-q implies not-p), contradiction (assume negation, derive absurdity), and case analysis. Verification tools automate routine steps; humans supply the creative ones - especially invariants.

### Key Points

- Induction: base case + inductive step proves all naturals.

- Structural induction handles recursive data structures.
- Strategies: direct, contrapositive, contradiction, cases.

### Key Terms

**Mathematical induction:** Proof over naturals via base and step.

**Structural induction:** Induction over recursive structures.

**Proof by contradiction:** Deriving absurdity from the negation.

### Example

- *Proving  $\sum_{i=1}^n i = n(n+1)/2$  by induction.*

**Summary:** Induction proves the unbounded; strategy chooses the path.

### Exam Tips

- ★ Very important: the induction schema.
- ★ Remember structural induction for data structures.

## Unit 2: Formal Specification

### Topic 6: What Is a Formal Specification

#### Learning Objectives

- Define formal specification
- Explain specification vs. implementation
- List specification qualities

#### Key Concepts

A formal specification is a precise, mathematical description of what a system must do - without saying how. It is a contract: implementers build to it, testers check against it, verifiers prove from it.

Good specifications are: unambiguous (one reading), complete (cover required behaviors), consistent (no contradictions), abstract (no implementation bias), and verifiable. Natural language fails the first; formal notations achieve it.

The specification process itself finds bugs: writing precisely exposes requirement gaps and contradictions before any code exists. Studies show specification-phase defect removal is the cheapest kind.

#### Key Points

- Specification: precise what, not how - a verifiable contract.
- Qualities: unambiguous, complete, consistent, abstract.
- Specifying itself exposes requirement defects early and cheaply.

#### Key Terms

**Formal specification:** A mathematical description of required behavior.

**Abstraction:** Omitting implementation detail.

**Consistency:** Freedom from contradictions.

#### Example

- Specifying 'sort' as: output is a permutation. and ordered - no algorithm mentioned.

**Summary:** Formal specification makes requirements precise enough to build and prove from.

### Exam Tips

- ★ Key point: specification qualities.
- ★ Remember what vs. how.

## Topic 7: Z Notation Basics

### Learning Objectives

- Read Z schemas
- Explain state and operations in Z
- Write simple Z specifications

### Key Concepts

Z (pronounced 'zed') specifies systems with set theory and predicate logic. A schema is a named box: declarations above the line, predicates below. State schemas declare variables with invariants; operation schemas relate before-state (undecorated) to after-state (primed, e.g.  $x'$ ).

Example - a counter: state has count: N; operations Increment ( $\text{count}' = \text{count} + 1$ ) and Reset ( $\text{count}' = 0$ ). Schema calculus composes schemas (conjunction, piping) into larger specifications.

Z's strength is data-rich systems (databases, information systems); its weakness is concurrency. Tool support (type checkers, animators) and the ISO standard matured it industrially.

### Key Points

- Z schemas: declarations + predicates; state vs. operation schemas.
- Primed variables denote after-state.
- Schema calculus composes specifications.

### Key Terms

**Schema:** Z's named specification unit.

**State schema:** Declares system state with invariants.

**Primed variable:** Denoting after-state value.

### Example

- Increment:  $\text{count}' = \text{count} + 1$

**Summary:** Z models data-rich systems with sets, logic, and schemas.

### Exam Tips

- ★ Key point: primed vs. unprimed meaning.
- ★ Remember schema structure.

## Topic 8: VDM and Model-Based Specification

### Learning Objectives

- Explain model-based specification
- Describe VDM constructs

- Compare VDM with Z

## Key Concepts

Model-based specification builds an abstract model of system state using mathematical data types (sets, sequences, maps), then defines operations as transformations. VDM (Vienna Development Method) is the classic method with its specification language VDM-SL.

VDM provides: basic types, composite types (records), invariants restricting valid states, and pre/postconditions on operations. Example: a stack specified with sequences, push/pop defined by sequence operations, invariant on capacity.

VDM vs. Z: similar philosophy; VDM emphasizes constructive data models and executable subsets (VDM++ for OO), Z emphasizes schema calculus. Both proved industrial value (IBM CICS, British Steel). Modern successors include Alloy and B.

## Key Points

- Model state with math types; define operations as transformations.
- VDM: types, invariants, pre/postconditions.
- Both Z and VDM proved industrial value.

## Key Terms

**Model-based specification:** Specifying via abstract mathematical models.

**Invariant:** A property always true of valid states.

**Pre/postcondition:** Requirements before/after an operation.

## Example

- *Stack pop: pre: stack nonempty; post: returns head. tail kept.*

**Summary:** Model-based methods specify data systems with mathematical precision.

## Exam Tips

- ★ Key point: invariants and pre/postconditions.
- ★ Remember model-based idea.

# Topic 9: Algebraic Specification

## Learning Objectives

- Explain algebraic specification
- Write simple algebraic specs
- Discuss initial algebra semantics

## Key Concepts

Algebraic specification defines data types by their operations and the equations (axioms) between them - no concrete representation. A stack: sorts (Stack, Elem), operations (push, pop, top, empty, isEmpty), axioms:  $\text{pop}(\text{push}(s,e)) = s$ ;  $\text{top}(\text{push}(s,e)) = e$ .

This is implementation-independent: any representation satisfying the axioms works. Equational reasoning proves properties; term rewriting executes specifications (OBJ, Maude).

Strengths: abstract data types, clean reasoning. Weaknesses: awkward for stateful/reactive systems. Used for language semantics, abstract datatypes, and protocol data formats.

## Key Points

- Specify types by operations + equational axioms.
- Implementation-independent; supports equational proofs.
- Best for abstract data types.

## Key Terms

**Algebraic specification:** Types defined by operations and axioms.

**Axiom:** An equation the operations must satisfy.

**Sort:** A type name in algebraic specs.

## Example

- $top(push(s.e)) = e$

**Summary:** Algebra captures data types as pure behavior.

## Exam Tips

- ★ Key point: the stack axioms.
- ★ Remember implementation independence.

# Topic 10: Specifying Concurrent Systems

## Learning Objectives

- Explain why concurrency needs special methods
- Introduce process algebras
- Describe temporal logic basics

## Key Concepts

Concurrent systems (interacting processes) defeat sequential reasoning: interleavings explode, and properties concern ongoing behavior (liveness: 'something good eventually happens'; safety: 'nothing bad happens').

Process algebras (CSP, CCS, pi-calculus) model processes as communicating entities with operators for sequencing, choice, and parallelism. They analyze deadlock, livelock, and equivalence (bisimulation).

Temporal logic reasons about time: LTL (linear - 'always', 'eventually' along paths), CTL (branching - 'for all paths', 'there exists a path'). Model checkers verify temporal properties automatically - the industrial breakthrough for hardware and protocols.

## Key Points

- Concurrency: interleaving explosion; safety vs. liveness properties.
- Process algebras model communicating processes.
- Temporal logic (LTL/CTL) expresses time-dependent properties.

## Key Terms

**Liveness:** 'Something good eventually happens.'

**Safety:** 'Nothing bad ever happens.'

**Model checking:** Automatic verification over state spaces.

### Example

- 'Every request is eventually answered' - a liveness property.

**Summary:** Concurrency needs process models and temporal reasoning.

### Exam Tips

- ★ Very important: safety vs. liveness.
- ★ Remember LTL vs. CTL.

## Unit 3: Program Verification

### Topic 11: Hoare Logic

#### Learning Objectives

- Explain Hoare triples
- Apply assignment and sequence rules
- Discuss soundness

#### Key Concepts

Hoare logic proves programs correct with triples  $\{P\} C \{Q\}$ : if  $P$  holds before command  $C$ ,  $Q$  holds after. Partial correctness (if  $C$  terminates,  $Q$  holds); total correctness adds termination.

Rules: assignment ( $\{P[e/x]\} x := e \{P\}$  - substitute backwards), sequence, conditional (prove both branches), while (needs an invariant). The consequence rule strengthens preconditions and weakens postconditions using logic.

Example: proving a swap or a summation loop correct. Hoare logic underlies tools like Dafny and Frama-C; its ideas (contracts) appear in Eiffel, SPARK Ada, and modern assertion checkers.

#### Key Points

- $\{P\} C \{Q\}$ :  $P$  before implies  $Q$  after.
- Assignment via backward substitution; loops need invariants.
- Basis of contract-based verification tools.

#### Key Terms

**Hoare triple:**  $\{P\} C \{Q\}$ : correctness assertion.

**Partial correctness:** Correctness assuming termination.

**Consequence rule:** Adjusting assertions via implication.

#### Example

- $\{x = 5\} x := x + 1 \{x = 6\}$

**Summary:** Hoare logic turns programs into provable mathematics.

#### Exam Tips

- ★ Very important: the Hoare triple meaning.
- ★ Remember the assignment axiom.

### Topic 12: Preconditions, Postconditions, Invariants

#### Learning Objectives

- Write contracts for functions
- Explain loop invariants
- Discuss Design by Contract

### Key Concepts

Contracts specify functions: preconditions (caller obligations, e.g.  $x > 0$ ), postconditions (implementation guarantees, e.g.  $\text{result} \geq 0$ ), and class invariants (always true between operations). Violations pinpoint blame: broken precondition = caller bug; broken postcondition = implementation bug.

Loop invariants are the key creative step: a property true before, during, and after each iteration, strong enough to imply the postcondition. Finding them is the art of verification - e.g. for summation, 'sum equals total of processed elements'.

Design by Contract (Eiffel, Meyer) builds these into languages; runtime assertion checking catches violations in testing; static verifiers prove them. Contracts document intent better than comments.

### Key Points

- Preconditions bind callers; postconditions bind implementations; invariants always hold.
- Loop invariants bridge iterations to correctness.
- Design by Contract makes contracts executable.

### Key Terms

**Precondition:** Requirement on entry to an operation.

**Postcondition:** Guarantee on exit.

**Loop invariant:** Property preserved by each iteration.

### Example

- *sort(x): pre  $x \geq 0$ ; post  $\text{result} * \text{result} \approx x$*

**Summary:** Contracts make expectations explicit and checkable.

### Exam Tips

- ★ Very important: loop invariant concept.
- ★ Remember blame assignment.

## Topic 13: Model Checking

### Learning Objectives

- Explain model checking
- Describe state-space explosion
- List combating techniques

### Key Concepts

Model checking automatically verifies finite-state systems against temporal logic properties: build the model, specify the property, and the checker explores all states - returning 'holds' or a counterexample trace. Counterexamples are gold: concrete bug scenarios.

The curse: state-space explosion - states grow exponentially with components. Combats: symbolic model checking (BDDs/SAT represent sets of states), partial-order reduction (skip equivalent interleavings), abstraction (verify simplified models), and bounded model checking (search to depth  $k$  -

finds shallow bugs fast).

Industrial wins: hardware verification (Intel, IBM), device drivers (SLAM/SDV), protocols. SPIN, NuSMV, and CBMC are classic tools.

### Key Points

- Automatic verification with counterexample traces.
- State explosion is the fundamental challenge.
- Symbolic, abstraction, and bounded techniques scale it.

### Key Terms

**State-space explosion:** Exponential growth of system states.

**Counterexample:** A violating execution trace.

**Bounded model checking:** Verification up to a depth bound.

### Example

- *SPIN found a deadlock in a protocol design before implementation.*

**Summary:** Model checking automates proof for finite systems, fighting explosion cleverly.

### Exam Tips

- ★ Very important: state explosion and counterexamples.
- ★ Remember symbolic techniques.

## Topic 14: Theorem Proving

### Learning Objectives

- Distinguish automated and interactive proving
- Describe proof assistants
- Discuss proof effort

### Key Concepts

Theorem provers verify infinite-state systems and deep mathematics. Automated provers (SAT/SMT solvers like Z3) decide fragments fully automatically - the engines inside many tools. Interactive provers (Coq, Isabelle, Lean) need human guidance through tactics, proving anything expressible - at high effort.

SMT solvers decide combinations of theories (arithmetic, arrays, bit-vectors) - powering program verifiers (Dafny, Frama-C) and symbolic execution. Humans write annotations; solvers discharge routine proof obligations.

Effort reality: seL4 took ~20 person-years for 10,000 lines. Proofs suit stable, critical cores - not rapidly-churning app code. But verified components anchor whole systems' trust.

### Key Points

- SMT solvers: automatic for decidable fragments.
- Interactive provers (Coq/Isabelle): anything, with human effort.
- High assurance costs person-years; apply to critical cores.

### Key Terms

**SMT solver:** Satisfiability modulo theories: e.g. Z3.

**Tactic:** A proof step in interactive provers.

**Proof obligation:** A formula the verifier must discharge.

### Example

- *Z3 automatically proves array-bounds safety from annotations.*

**Summary:** Provers trade automation against expressiveness; effort focuses on critical cores.

### Exam Tips

- ★ Key point: SMT vs. interactive.
- ★ Remember seL4's effort scale.

## Topic 15: Refinement: From Spec to Code

### Learning Objectives

- Explain refinement
- Describe refinement steps
- Discuss the B method

### Key Concepts

Refinement develops implementations from specifications through correctness-preserving steps: start abstract, add detail, prove each step preserves behavior. The final code is correct by construction.

Refinement relations: data refinement (abstract sets become arrays), operation refinement (spec operations become algorithms). Proof obligations show concrete steps simulate abstract ones.

The B method (Abrial) industrialized this: abstract machines with invariants, refined through layers to code - famously used for Paris Metro Line 14's driverless trains. Event-B extends B to reactive systems. Refinement suits systems specified completely upfront.

### Key Points

- Refinement: stepwise, provable development from spec to code.
- Data and operation refinement add detail preserving behavior.
- B/Event-B industrialized it (Paris Metro).

### Key Terms

**Refinement:** Correctness-preserving development steps.

**Data refinement:** Replacing abstract data with concrete representations.

**B method:** Abrial's refinement-based formal method.

### Example

- *A metro signaling spec refined to executable controllers. proved safe.*

**Summary:** Refinement builds correctness in rather than testing it in.

### Exam Tips

- ★ Key point: refinement idea.
- ★ Remember the Paris Metro example.

## Unit 4: Applications and Practice

### Topic 16: Verifying Real Software: Case Studies

#### Learning Objectives

- Survey landmark verifications
- Extract lessons
- Assess current capabilities

#### Key Concepts

Landmarks: seL4 microkernel (functional correctness + security), CompCert C compiler (preserves program semantics), Paris Metro signaling (B method, zero defects), AWS's use of TLA+ (found deep S3 bugs pre-deployment), and verified cryptography (HACL\* in Firefox).

Lessons: verify stable critical components; specifications are the hard part (writing what 'correct' means); tool maturity now supports real code (C, Rust); Amazon shows lightweight formal methods (TLA+ modeling) catch design bugs cheaply.

Current frontier: verified OS components, blockchain contracts (high-value bugs!), and AI system properties. Formal methods moved from academic to industrial over two decades.

#### Key Points

- seL4, CompCert, Paris Metro, AWS TLA+: proof at scale works.
- Specs are the hard part; tools now handle real code.
- Lightweight methods (TLA+) catch design bugs cheaply.

#### Key Terms

**TLA+:** Lamport's specification language for concurrent systems.

**CompCert:** A verified optimizing C compiler.

**Smart contract:** Blockchain code where bugs cost millions.

#### Example

- *AWS found a subtle S3 consistency bug with TLA+ before coding.*

**Summary:** Formal methods now protect real systems industrially.

#### Exam Tips

- ★ Key point: two case studies and their lessons.
- ★ Remember TLA+ at AWS.

### Topic 17: Alloy and Lightweight Modeling

#### Learning Objectives

- Explain lightweight formal methods
- Describe Alloy basics
- Discuss scope-bounded analysis

#### Key Concepts

Lightweight formal methods favor automation and partial analysis over full proof - 'the small-scope hypothesis': most bugs have small counterexamples, so checking all small cases finds most bugs.

Alloy (Jackson, MIT) models structures with signatures (sets) and relations, constraints as formulas; the Analyzer searches bounded scopes for instances or counterexamples automatically. Users iterate: model, check, fix - a design sketchpad with teeth.

Uses: access-control policies, network configurations, data models. Alloy's genius is accessibility - designers without proof training find real flaws in hours. Complements heavyweight verification.

### Key Points

- Small-scope hypothesis: small counterexamples reveal most bugs.
- Alloy: signatures, relations, automatic bounded analysis.
- Lightweight methods suit design-phase flaw-finding.

### Key Terms

**Alloy:** A lightweight modeling language with automatic analysis.

**Small-scope hypothesis:** Most flaws appear in small instances.

**Bounded analysis:** Exhaustive checking within size limits.

### Example

- *Modeling a permission scheme in Alloy reveals an unintended admin loophole.*

**Summary:** Lightweight modeling finds design flaws fast and cheaply.

### Exam Tips

- ★ Very important: the small-scope hypothesis.
- ★ Remember Alloy's approach.

## Topic 18: Tools Tour: What Exists

### Learning Objectives

- Survey major tool categories
- Match tools to tasks
- Discuss adoption barriers

### Key Concepts

Categories: specification languages (Z, VDM, Alloy, TLA+), model checkers (SPIN, NuSMV, CBMC), theorem provers (Coq, Isabelle, Lean), SMT solvers (Z3, CVC5), program verifiers (Dafny, Frama-C, VeriFast), and static analyzers (Coverity, Infer - lightweight, industrial).

Matching: design flaws -> Alloy/TLA+; protocol/hardware -> model checkers; critical code -> Dafny/Frama-C; deep math -> Coq/Isabelle; routine bug-finding -> static analyzers in CI.

Barriers: learning curves, annotation burden, scalability limits, and cultural resistance ('we test'). Adoption grows via education, better automation, and success stories - plus curricula like this one.

### Key Points

- Tool per task: Alloy/TLA+ (design), SPIN (protocols), Dafny (code), Z3 (engine).
- Static analyzers bring lightweight value to CI pipelines.
- Barriers: learning, annotation, scale, culture.

## Key Terms

**Static analyzer:** Tool finding bugs without executing code.

**Program verifier:** Tool proving code against specifications.

**CI:** Continuous integration.

## Example

- *Infer runs on every pull request, catching null dereferences.*

**Summary:** A rich toolchain exists; matching tool to task is the skill.

## Exam Tips

- ★ Key point: match three tools to tasks.
- ★ Remember adoption barriers.

# Topic 19: Formal Methods in Agile and DevOps

## Learning Objectives

- Discuss integrating formality with agility
- Explain continuous verification
- Describe pragmatic adoption

## Key Concepts

Formal methods need not oppose agility: specify critical components formally while iterating the rest; use lightweight modeling in design sprints; run verifiers/analyzers in CI pipelines (continuous verification) - proofs as automated tests.

Pragmatic adoption ladders: level 1 - write precise pre/postconditions as documentation; level 2 - runtime assertion checking; level 3 - static analyzers in CI; level 4 - targeted verification of critical modules; level 5 - full refinement. Climb as value justifies.

Case: Amazon's teams write TLA+ specs for new services in days, catching design bugs before coding - formal methods at startup speed. The lesson: fit the method's weight to the risk.

## Key Points

- Formality scales: document, assert, analyze, verify - climb as justified.
- Continuous verification: proofs in CI pipelines.
- Amazon's TLA+ shows lightweight formality at speed.

## Key Terms

**Continuous verification:** Automated proof/analysis in delivery pipelines.

**Assertion checking:** Runtime validation of contracts.

**Pragmatic adoption:** Graduated formality matched to risk.

## Example

- *A team adds Dafny verification only to its crypto module.*

**Summary:** Agile teams adopt formality incrementally, continuously, and where it pays.

## Exam Tips

- ★ Key point: the adoption ladder.

★ Remember continuous verification. 

## Topic 20: Limits, Criticisms, and the Future

### Learning Objectives

- State honest limitations
- Address classic criticisms
- Identify future directions

### Key Concepts

Honest limits: verification proves the model, not reality - wrong specs prove wrong systems; the environment (hardware faults, human operators) escapes models; effort remains high; and most software's value doesn't justify proof.

Classic criticisms: 'it doesn't scale' (less true yearly), 'it ignores the real problem' (requirements), 'mathematicians only' (tools improve). Responses: apply where valuable, combine with testing, and keep improving usability.

Future: verified AI/ML properties, quantum-safe cryptography proofs, proof automation via AI (learning tactics, suggesting invariants), and verified infrastructure as code. The trajectory: more automation, broader adoption, deeper integration with engineering practice.

### Key Points

- Limits: model-reality gap, environment, effort, applicability.
- Answer criticisms with targeted use and better tools.
- Future: AI-assisted proving, verified ML, broader automation.

### Key Terms

**Model-reality gap:** Difference between verified model and deployed system.

**Proof automation:** AI/ML assisting proof construction.

**Verified infrastructure:** Proved-correct deployment configurations.

### Example

- *A proved-correct controller still fails if its sensor lies - environment matters.*

**Summary:** Formal methods are powerful, bounded tools with an expanding future.

### Exam Tips

- ★ Key point: the model-reality gap.
- ★ Remember honest limitations.

## Glossary of Key Terms

**Formal method:** A mathematically-based specification or verification technique.

**Hoare triple:**  $\{P\} C \{Q\}$ : a correctness assertion about a command.

**Loop invariant:** A property preserved by each loop iteration.

**Model checking:** Automatic verification over finite state spaces.

**Temporal logic:** Logic reasoning about time-dependent properties.

**Refinement:** Correctness-preserving development from spec to code.

**Z notation:** A schema-based formal specification language.

**Alloy:** A lightweight modeling language with automatic analysis.

**SMT solver:** An automatic prover for decidable theories, e.g. Z3.

**Safety property:** 'Nothing bad ever happens.'

**Liveness property:** 'Something good eventually happens.'

**Precondition:** A requirement before an operation executes.

**Postcondition:** A guarantee after an operation completes.

**State-space explosion:** Exponential growth of verification state spaces.

**TLA+:** Lamport's language for specifying concurrent systems.