BUILD CORRECT SOFTWARE

FORMAL VERIFICATION FOR WORKING SOFTWARE ENGINEERS

FROM FIRST PRINCIPLES TO PRODUCTION-GRADE VERIFIED SYSTEMS

EXPRESS PRECISE PROPERTIES
Turn requirements into machine-checkable specifications

CHOOSE THE RIGHT TECHNIQUE
Select the best verification approach for each problem

INTERPRET RESULTS
Understand solver outcomes and counterexamples with confidence

BUILD PROOFS
Use modern tools to develop and maintain reliable proofs

INTEGRATE INTO WORKFLOWS
Introduce verification incrementally into real development processes

MIKE RAFAEL

Formal Verification for Working Software Engineers

From First Principles to Production-Grade Verified Systems

Steve Publications

This book is available at
https://leanpub.com/formalverificationforworkingsoftwareengineers

This version was published on 2026-08-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

From First Principles to Production-Grade Verified Systems

About this book: This is a practical guide to formal verification written for software engineers who know how to write production code but have never used formal methods. You will learn to express precise properties about your software, choose the right verification technique for each problem, and integrate mathematical guarantees into real development workflows. By the end, you will be able to identify high-value verification targets, construct machine-checkable specifications, interpret solver results and counterexamples, develop proofs using modern tools, and progressively introduce formal verification into systems ranging from ordinary application code to concurrent, distributed, and security-critical software.

Introduction: Why Testing Is Not Enough

A History of Missed Bugs

On a humid morning in June 1985 at a radiation therapy clinic in Marietta, Georgia, a cancer patient named Katie Yarbrough lay on the treatment table for her twenty-fourth session. The machine above her was a Therac-25, a state-of-the-art computer-controlled radiotherapy device built by Atomic Energy of Canada Limited. It was supposed to deliver 200 rads of radiation. Instead it delivered somewhere between fifteen thousand and twenty thousand rads, roughly one hundred times the intended dose. Yarbrough screamed in agony during treatment. Over the following months she endured a mastectomy and permanent disfigurement. She survived, but barely [1].

This was not an isolated incident. Between 1985 and 1987 at least six patients received massive overdoses from Therac-25 machines. Three of them died. The events were described by investigators as the worst series of radiation accidents in thirty-five years of medical accelerator history [2].

What went wrong? A combination of failures, each plausible on its own and together catastrophic. The machine had removed hardware safety interlocks that existed on earlier models, relying instead on software checks. A race condition in the control software allowed a high-current electron beam to fire without the tungsten target properly positioned when operators rapidly switched between treatment modes within eight seconds. An integer overflow caused safety flags to reset to zero, bypassing protective checks. Error messages were cryptic and unhelpful, displaying codes like Malfunction 54 that gave operators no indication of danger. When errors appeared, technicians often restarted the machine and continued treatment [3].

The Therac-25 story is notorious in software engineering for a reason: it shows how bugs that seem obvious in hindsight can slip through every conventional quality gate when testing alone is relied upon. The race condition required a very specific sequence of rapid operator keystrokes under precise

timing conditions. The integer overflow only manifested when particular internal counters reached unusual values. Neither would have appeared in typical usage or standard test suites.

A decade later another famous failure demonstrated a different kind of testing blind spot. On June 4, 1996 the Ariane 5 rocket, Europe's newest launch vehicle carrying a payload worth over three hundred million dollars, exploded thirty-seven seconds after liftoff during its maiden flight [4]. The cause was an integer overflow in guidance software that had been reused from the earlier Ariane 4 rocket. On Ariane 4 the horizontal velocity values never exceeded what could fit in a sixteen-bit signed integer. On Ariane 5 they did. The conversion from sixty-four-bit float to sixteen-bit integer overflowed, crashing the inertial reference system and triggering full nozzle deflection that destroyed the vehicle.

The software had been tested extensively on Ariane 4 where it worked perfectly. It had never been tested against Ariane 5 flight trajectories because engineers assumed reuse implied correctness. The assumption was wrong, and the consequences were spectacular.

These are not edge cases in the history of software engineering. They are examples of a fundamental limitation: testing can only show the presence of bugs, never their absence. Edsger Dijkstra famously stated this in 1970, and it remains true today [5]. No matter how many test cases you write, you have verified behavior on a finite set of inputs against an infinite space of possibilities. The bug that matters is always the one your tests did not exercise.

What Formal Verification Actually Means

Formal verification is the practice of using mathematics to prove or disprove that a system satisfies its specification under all possible conditions. It is not testing with better tools. It is not static analysis with fewer false positives. It is a fundamentally different approach: instead of checking whether a program behaves correctly on selected inputs, you prove that it must behave correctly on every input for which the specification applies.

This definition requires unpacking three terms: formal, verification, and specification.

Formal means expressed in a precise mathematical language with well-defined syntax and semantics. There is no ambiguity, no hand-waving, no

natural language vagueness. When you write a formal specification, every symbol has an exact meaning defined by the logic you are using.

Verification means establishing truth through rigorous reasoning rather than empirical observation. In testing you observe behavior on concrete executions. In verification you construct a logical argument that covers all possible executions satisfying certain conditions. The result is a guarantee, not a probability.

Specification means a precise statement of what the system should do. This is often the hardest part. A formal proof can only establish that your code matches your specification. If the specification is wrong or incomplete, the proof is correct but useless. We will return to this point repeatedly because it is where many verification efforts fail in practice.

Formal verification sits at one end of a spectrum of assurance techniques. At the other end is manual testing with hand-crafted test cases. Between them lie automated testing, property-based testing, fuzzing, static analysis, and various levels of formal methods. Each technique provides different kinds of guarantees at different costs. The goal of this book is to help you understand where formal verification fits in that spectrum, when it is worth the investment, and how to use it effectively.

There are two main families of formal verification techniques. Model checking exhaustively explores all reachable states of a finite model to verify whether properties hold. It is fully automatic: you provide a model and properties, and the tool either proves them or produces a counterexample showing exactly where they fail. The limitation is scalability: state spaces grow exponentially with system size, so model checking works best on abstracted models rather than full implementations.

Deductive verification uses logical reasoning to prove that code satisfies specifications. You annotate your program with preconditions, postconditions, and invariants, then a verification tool generates logical formulas called verification conditions that must be true for the program to be correct. These conditions are checked either automatically by SMT solvers or interactively using proof assistants. Deductive verification can handle much larger systems than model checking but requires more expertise and manual effort from the engineer.

Both approaches share a crucial property: they are exhaustive within their domain. Model checking explores every reachable state of the modeled

system. Deductive verification proves properties for all inputs satisfying the precondition. Neither depends on selecting representative test cases or hoping that coverage metrics indicate thoroughness.

The Spectrum from Testing to Proofs

Understanding where formal verification fits requires understanding what comes before it. Let me walk through the main techniques engineers use to gain confidence in their software, from weakest to strongest guarantees.

Manual testing with hand-crafted test cases is the most common approach. You write tests that exercise expected behavior and some anticipated edge cases. This catches obvious bugs early but provides essentially no guarantee about untested paths. A thousand passing tests still leaves infinite untested inputs.

Automated testing scales manual testing by running large suites repeatedly, typically in CI pipelines. More tests mean more coverage and faster feedback on regressions, but the fundamental limitation remains: you are still checking finite examples against infinite possibilities.

Property-based testing changes the approach by asking you to specify properties that should hold for all inputs rather than writing individual test cases. A property might state that sorting any list produces a permutation of the original elements in non-decreasing order. The testing framework then generates hundreds or thousands of random inputs and checks whether the property holds on each one. This is more powerful than example-based testing because it explores input space systematically and can discover surprising edge cases you never thought to test. But it is still testing: if no generated input triggers a bug, the property passes even though a counterexample might exist.

Fuzzing takes an adversarial approach by feeding programs with random or mutated inputs designed to trigger crashes or unexpected behavior. Modern fuzzers use coverage feedback to guide mutation toward unexplored code paths. Fuzzing is excellent at finding memory safety bugs, parsing errors, and other robustness issues. Like all testing it provides no guarantee of completeness.

Static analysis examines code without executing it to detect patterns associated with bugs: null pointer dereferences, buffer overflows, data races, unreachable code. Modern static analyzers use sophisticated techniques

including abstract interpretation to reason about program behavior soundly. Some static analyses are complete for specific bug classes and can prove absence of those bugs under defined assumptions. Others are heuristic and produce false positives or miss real issues. Static analysis is increasingly formal in nature but often falls short of full verification because it must balance precision against scalability.

Model checking, as described above, provides exhaustive guarantees about finite models. If a model checker proves a property holds, it holds for all behaviors of the model. The catch is that you must trust the model accurately represents the real system, and the state space must be small enough to explore completely.

Deductive verification provides the strongest practical guarantees: mathematical proofs that code satisfies its specification under stated assumptions. These proofs are machine-checked so they cannot contain human reasoning errors that slip through peer review. The trade-offs are increased upfront cost in writing specifications and potentially more expertise required to complete proofs for complex properties.

The key insight is that these techniques are complementary, not competing. A mature verification strategy uses testing for broad coverage of functional behavior, fuzzing for robustness against malformed inputs, static analysis for systematic bug pattern detection, and formal verification for the most critical properties where guarantees matter more than cost. We will explore how to combine them effectively in Chapter 16.

How This Book Is Structured

This book is organized as a progressive journey from foundations to advanced practice. Each chapter builds on previous material so you can read sequentially and develop understanding incrementally. However once you are comfortable with the basics you may want to jump between chapters based on your immediate needs.

The Introduction establishes why testing alone is insufficient and defines what formal verification means in practical terms.

Chapter 1 introduces properties, specifications, and invariants: the fundamental language of verification. You will learn to express preconditions, postconditions, and invariants that capture what your code should guarantee.

Chapter 2 covers the mathematical foundations needed for verification: propositional logic, predicate logic, sets, relations, functions, and induction. These are presented with direct connections to programming rather than as abstract mathematics.

Chapter 3 introduces Hoare logic, the foundational framework for reasoning about imperative programs. You will learn how to construct correctness proofs for simple programs using inference rules and weakest preconditions.

Chapter 4 uses property-based testing as a bridge between conventional testing and formal verification. This is an accessible entry point that introduces formal thinking without requiring proofs.

Chapters 5 through 9 cover the core automated verification techniques: SAT solving, SMT solving, symbolic execution, abstract interpretation, and model checking with temporal logic. Each chapter explains how the technique works, what guarantees it provides, its limitations, and representative tools.

Chapter 10 introduces deductive verification with contracts and annotations using Dafny as the primary vehicle. This is where you start verifying real programs with machine-checked proofs.

Chapter 11 covers separation logic for reasoning about mutable state and memory, essential for systems programming verification.

Chapters 12 and 13 address concurrent and distributed system verification: linearizability, race freedom, consensus protocols, and failure handling.

Chapter 14 introduces interactive theorem proving with Lean, Coq/Rocq, and Isabelle/HOL for the most demanding verification tasks.

Chapter 15 surveys verified systems at scale: compilers, operating system kernels, cryptographic implementations, and what we can learn from million-line verified projects.

Chapter 16 focuses on engineering practice: how to introduce formal verification into real teams, integrate it with CI/CD pipelines, choose what to verify, and maintain proofs as software evolves.

The Major Case Study walks through progressive strengthening of guarantees on a realistic component, showing the evolution from conventional tests to machine-checked proofs with all the practical challenges along the way.

Throughout the book you will find annotated code examples, proof sketches, tool walkthroughs, and discussions of real-world trade-offs. The

goal is not to turn you into a logician but to give you practical capability to use formal methods where they matter most in your work.

Let us begin.

Chapter 1: Properties, Specifications, and Invariants

Before you can verify anything, you must be able to say precisely what correct behavior means. This chapter introduces the fundamental concepts that form the language of verification: properties, specifications, preconditions, postconditions, invariants, assertions, and state machines. These ideas are not unique to formal methods; they appear in good software design regardless of whether you use formal tools. But understanding them explicitly is essential for verification because every proof ultimately establishes that some property holds.

From Requirements to Precise Properties

Software requirements are typically written in natural language: the system shall authenticate users before granting access, transactions must be atomic, the response time shall not exceed two seconds under normal load. These statements capture intent but lack precision. What does atomic mean for a distributed transaction? Under what conditions must the response time bound hold? What happens if authentication fails?

Verification requires properties that are unambiguous and machine-checkable. A property is a statement about program behavior that can be evaluated as true or false. The transition from requirements to properties involves making implicit assumptions explicit, quantifying over cases, and eliminating natural language vagueness.

Consider this requirement for a banking system: transfers between accounts must preserve the total balance. As stated this is vague. Does it mean before and after each transfer? What about concurrent transfers? What if one account has insufficient funds? A precise property might be:

For any sequence of completed transfers, the sum of all account balances after execution equals the sum of all account balances before execution began.

This property quantifies over all sequences of transfers and all accounts. It applies only to completed transfers, not ones that failed or are in progress. It makes a concrete claim about sums that can be checked mathematically. Notice also that this property does not specify how transfers work internally: it constrains observable behavior without prescribing implementation. This separation between what must hold and how to achieve it is central to good specification design.

Not every requirement translates cleanly into a verifiable property. Performance requirements like response time bounds depend on environmental factors beyond the software's control: network latency, hardware capacity, workload patterns. You can verify that an algorithm has certain asymptotic complexity, but proving concrete timing guarantees requires modeling the execution environment precisely, which is often impractical. Safety properties are generally easier to verify than liveness or performance properties because they state what must never happen rather than what must eventually happen.

Preconditions and Postconditions: Contracts for Code

The most basic verification unit is a single function or method. To reason about whether a function is correct, you need to specify two things: what the caller must guarantee before calling it, and what the function guarantees after it returns. These are called preconditions and postconditions.

A precondition is a condition that must hold before a function executes for the function to behave correctly. If the precondition is violated, the function makes no promises about its behavior: it may crash, return garbage, or silently corrupt state. The responsibility for satisfying preconditions lies with the caller.

A postcondition is a condition that the function guarantees will hold after it returns, provided the precondition was satisfied and the function terminates normally. If the postcondition does not hold when the function returns, there is a bug in the implementation.

Together these form a contract between caller and callee, analogous to Bertrand Meyer's design by contract methodology [1]. The contract clarifies responsibility: if something goes wrong, either the caller violated a precondition or the callee failed to establish its postcondition. There is no ambiguity about who is at fault.

Let us look at a concrete example. Consider a function that divides two integers:

```
function divide(a: int, b: int) -> int:
    return a / b
```

What should the contract be? A natural precondition is b != 0, because division by zero is undefined. The postcondition might state that the result equals the mathematical quotient of a divided by b. But integer division truncates, so we need to be precise about what quotient means. A correct postcondition for truncated integer division would be:

result * b <= a < (result + 1) * b when b > 0 or equivalently for negative divisors with appropriate sign handling.

This is more complex than the naive result == a / b because it precisely characterizes truncation behavior rather than relying on informal understanding of what the division operator does.

Preconditions and postconditions serve multiple purposes beyond formal verification. They document expected usage for other developers, enable static analysis tools to detect misuse, can be checked at runtime during development to catch bugs early, and form the basis for automated verification when combined with proof tools. Many modern languages support contracts natively or through libraries: Ada has pre/post/invariant clauses, Eiffel was designed around contracts from the start, Java supports them via annotations like JSR-305 and Checker Framework, Rust uses assert! macros and Result types to encode similar ideas.

Invariants: What Must Always Be True

An invariant is a property that must hold at all times during program execution, or more precisely at all observable points in execution. Unlike preconditions and postconditions which apply at specific boundaries (function entry and exit), invariants constrain the entire lifetime of some entity: an object, a data structure, or the whole system.

Object invariants describe conditions that must hold for every valid instance of a class. For example, a Stack class might maintain the invariant that its size field always equals the length of its internal array of elements. If this invariant

is violated, the stack is corrupted and all operations on it become unreliable. The responsibility for maintaining object invariants lies with the object itself: every method must ensure that if the invariant held before the method was called, it still holds afterward.

Data structure invariants are similar but often more complex. A binary search tree maintains the invariant that for every node, all keys in its left subtree are less than the node's key and all keys in its right subtree are greater. This invariant is what makes search efficient: without it the structure is just a random collection of nodes. Any operation that modifies the tree must preserve this invariant, which often requires careful restructuring like rotations in balanced trees.

System invariants apply to an entire system rather than individual components. The banking example from earlier describes a system invariant: total balance is preserved across transfers. A concurrent data structure might maintain the invariant that at most one thread holds a particular lock at any time. Distributed systems often have invariants about consistency: all replicas eventually agree on the same value, or a certain quorum of nodes must acknowledge a write before it is considered committed.

Invariants are crucial for verification because they enable modular reasoning. If you can prove that each operation preserves the invariant, and the invariant implies whatever safety property you care about, then you know the property holds throughout execution without having to reason about every possible sequence of operations. This compositional approach is what makes verifying large systems tractable: you verify local properties of components and derive global properties from how they compose.

The key challenge with invariants is finding them. A good invariant must be strong enough to imply the desired safety properties but weak enough that it actually holds throughout execution. Too strong and no implementation can satisfy it; too weak and it provides no useful guarantees. Discovering appropriate invariants requires understanding both what you want to guarantee and how your code works internally.

Assertions as Lightweight Verification

An assertion is a Boolean expression embedded in code that claims something must be true at that point during execution. If the assertion evaluates to false,

the program has violated some expected property and typically halts with an error message. Assertions are the simplest form of formal specification: they express properties precisely enough to be checked mechanically but informally enough to write without deep logical training.

Assertions serve several purposes. They document assumptions about program state that other developers should understand. They catch bugs early during development by detecting violated assumptions before they propagate and cause confusing failures elsewhere. They can guide reasoning about correctness by making implicit expectations explicit. And in verification contexts, assertions become properties that must be proven rather than merely checked at runtime.

Consider this function that searches for an element in a sorted array using binary search:

```
function binarySearch(arr: int[], target: int) -> int:
    low = 0
    high = arr.length - 1

    assert arr is sorted ascendingly  // precondition
    assert low <= high + 1           // invariant candidate

    while low <= high:
        mid = low + (high - low) / 2

        if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

        assert low <= high + 1       // loop invariant check

    return -1
```

The assertions here express key properties. The first is a precondition: binary search only works on sorted arrays, and if someone calls it on an unsorted array the results are meaningless. The second and third assertions are candidate loop invariants that should hold before each iteration. These help verify that the search narrows correctly without skipping the target.

In practice assertions are typically enabled during development and testing but disabled in production to avoid performance overhead. This means they

cannot be relied upon for safety in deployed systems: a bug that only manifests when assertions are off will go undetected. However assertions are excellent for catching bugs during development and forming the basis for more rigorous verification later. If you can prove an assertion always holds using formal methods, you have established a real guarantee rather than hoping tests exercise the right paths.

State Machines and Behavioral Models

Many systems are best understood not as collections of functions but as state machines: entities that exist in one of a finite set of states and transition between states in response to events or actions. A traffic light cycles through green, yellow, and red. A database connection moves from disconnected to connecting to connected to closing to closed. An authentication system tracks whether a user is logged in, what permissions they have, and when their session expires.

A state machine model consists of:

- A set of states representing all possible configurations
- A set of transitions specifying how the system moves between states
- An initial state or set of valid initial states
- Optionally, actions associated with transitions (what happens when a transition occurs)

State machines are powerful for verification because they make behavior explicit and finite. You can reason about what sequences of transitions are possible, whether certain states are reachable, and whether the system can enter undesirable states like deadlocks or inconsistent configurations.

Consider a simplified model of a file lock:

States: Unlocked, LockedBy(client_id) Transitions:

- Unlocked −lock(c)−> LockedBy(c)
- LockedBy(c) −unlock(c)−> Unlocked
- LockedBy(c) −unlock(d) where d != c −> ERROR (wrong client unlocking)

From this model you can state and verify properties like:

- The file is never locked by two different clients simultaneously
- Every lock operation on an unlocked file succeeds
- An unlock operation by the wrong client is always detected

These are safety properties that should hold for any correct implementation of file locking. By verifying them against the model first, before writing code, you can catch design flaws early and have a precise specification to verify the implementation against later.

State machine modeling is particularly valuable for concurrent and distributed systems where interaction between components creates complex behavior hard to reason about from code alone. Tools like TLA+ are designed specifically for specifying and verifying state machines at the system level. We will explore TLA+ in detail in Chapter 9.

Designing Specifications That Are Checkable

The hardest part of verification is often not the proving but the specifying. A specification that is too vague cannot be checked. One that is too detailed becomes a restatement of the implementation and provides no independent guarantee. Finding the right level of abstraction requires judgment and practice.

Here are principles that help:

First, separate what from how. A good specification describes observable behavior without constraining implementation details. Specify that a sorting function produces a sorted permutation of its input, not that it uses quicksort or mergesort. This gives implementers freedom to choose efficient algorithms while maintaining a clear correctness criterion.

Second, quantify explicitly. Natural language requirements often leave quantifiers implicit: users shall be authenticated before accessing resources. Does this mean all users? All accesses? Under all conditions? A precise specification states for all users u and all resource access requests r by u, if u is not authenticated then r is denied. Explicit quantification eliminates ambiguity about scope.

Third, handle edge cases deliberately. What happens with empty inputs, maximum values, concurrent operations, failures? Specifications that ignore

edge cases are incomplete: they provide no guarantee about behavior in those situations, which means the implementation can do anything without violating the spec. Either specify expected behavior for edge cases or explicitly exclude them from the specification's scope.

Fourth, make assumptions explicit. Every verification rests on assumptions about the environment: memory is available, the clock moves forward, network messages eventually arrive (or do not). If you verify a protocol assuming reliable message delivery but deploy it on an unreliable network, the proof is correct but irrelevant. State your assumptions clearly so they can be evaluated independently of the proof.

Fifth, keep specifications testable where possible. A specification should ideally be precise enough that you can write tests or property-based checks against it even before doing formal verification. If you cannot operationalize a property in tests, it is probably too vague to verify formally either. Specifications that guide both testing and proof are more likely to capture real intent.

The art of specification design improves with practice. As you work through examples in this book you will develop intuition for what makes a good specification: precise enough to be checkable, abstract enough to be useful, complete enough to cover important cases, and grounded in real requirements rather than mathematical elegance alone.

Summary

This chapter introduced the fundamental concepts of verification as properties and specifications. You learned that preconditions state what callers must guarantee, postconditions state what functions guarantee in return, and invariants constrain valid states throughout execution. Assertions provide lightweight runtime checks that can evolve into formal properties. State machines model system behavior explicitly for reasoning about sequences of events. And good specifications balance precision with abstraction to be both checkable and useful.

These concepts form the vocabulary you will use throughout the rest of this book. Every verification technique, from property-based testing to interactive theorem proving, ultimately establishes that some property holds under some conditions. Understanding what properties matter and how to state them precisely is more important than mastering any particular tool.

Chapter 2: Logic and Mathematics for Verification

Formal verification rests on logic and mathematics, but you do not need a math degree to use it effectively. This chapter presents the essential logical and mathematical concepts needed for verification, explained with direct connections to programming. We will cover propositional logic, predicate logic, sets and relations, functions, induction, and basic proof techniques. The goal is not mathematical sophistication but practical fluency: enough understanding to read specifications, write properties, and follow verification arguments without getting lost in notation.

Propositional Logic: Truth Tables and Equivalences

Propositional logic is the simplest form of formal logic. It deals with propositions: statements that are either true or false. Examples include x > 0, the user is authenticated, or the array is sorted. Each proposition has a definite truth value even if we do not know what it is.

The building blocks of propositional logic are logical connectives that combine propositions into more complex ones. The five standard connectives are:

NOT (negation): written as not P or ¬P. This is true when P is false and false when P is true. In code this corresponds to the ! operator.

AND (conjunction): written as P and Q or P ▫ Q. True only when both P and Q are true. Corresponds to && in most languages.

OR (disjunction): written as P or Q or P ▫ Q. True when at least one of P or Q is true. Note this is inclusive: P or Q is true even if both are true. This corresponds to || in most languages.

IMPLIES (implication): written as P implies Q or P ▫ Q. This is the trickiest connective for beginners. The statement P ▫ Q means if P is true then Q must be true. It is false only when P is true and Q is false. When P is false, P ▫

Q is automatically true regardless of Q. This may seem counterintuitive but it captures the idea that an implication makes no claim about cases where the premise does not hold.

IF AND ONLY IF (biconditional): written as P iff Q or P ↔ Q. True when P and Q have the same truth value: both true or both false. This states that P and Q are logically equivalent.

Truth tables systematically show how connectives evaluate for all combinations of input truth values. Here is the truth table for implication:

P | Q | P □ Q T | T | T T | F | F F | T | T F | F | T

The only false case is when the premise P holds but the conclusion Q does not. This matches our intuitive understanding of conditional guarantees: a promise is broken only when the condition for the promise is met but the promised outcome does not occur.

Understanding implication correctly is crucial for verification because preconditions and postconditions are expressed as implications. A function contract states that if the precondition holds before calling, then the postcondition will hold after returning. Symbolically this is precondition □ postcondition. If the precondition does not hold, the contract makes no claim about the result: the implication is vacuously true.

Logical equivalences are pairs of formulas that always have the same truth value regardless of their components. Some important equivalences include:

De Morgan's laws: not (P and Q) is equivalent to (not P) or (not Q). Similarly not (P or Q) is equivalent to (not P) and (not Q). These are used constantly when simplifying negated conditions.

Contrapositive: P □ Q is equivalent to (not Q) □ (not P). This equivalence underlies proof by contrapositive: to prove if P then Q, you can instead prove if not Q then not P. Sometimes the contrapositive form is easier to reason about.

Double negation: not (not P) is equivalent to P. Negating twice cancels out.

These equivalences are not just academic curiosities. They appear constantly in verification when simplifying conditions, understanding what a negated property means, or transforming specifications into forms that solvers can handle more easily.

Predicate Logic: Quantifiers and Properties Over Data

Propositional logic is limited because it cannot reason about properties of individual elements or make claims about all elements in a collection. Predicate logic extends propositional logic with quantifiers that enable reasoning over domains of values.

A predicate is a property that depends on one or more variables. For example, isPrime(n) is true when n is prime and false otherwise. The expression isPrime(x) is not itself true or false until x is given a specific value. Predicates are the logical equivalent of functions that return Boolean values.

The universal quantifier □ (for all) states that a property holds for every element in a domain. The formula □x. P(x) reads for all x, P(x) holds. For example:

□i. 0 <= i < arr.length □ arr[i] >= 0

This states that every element of the array is non-negative. To prove this universally quantified statement you must show the property holds for an arbitrary index i, not just specific indices you checked. This distinction between checking examples and proving universality is fundamental to verification.

The existential quantifier □ (there exists) states that at least one element satisfies a property. The formula □x. P(x) reads there exists an x such that P(x). For example:

□i. 0 <= i < arr.length □ arr[i] == target

This states that the target value appears somewhere in the array. To prove this you need only exhibit one specific index where the property holds.

Quantifiers interact with logical connectives in important ways. The negation of a universal quantifier becomes an existential quantifier with negated body: not (□x. P(x)) is equivalent to □x. not P(x). If it is not true that all elements satisfy P, then there must be at least one that does not. Similarly not (□x. P(x)) is equivalent to □x. not P(x). These equivalences are critical when interpreting counterexamples: a model checker proving that not all states satisfy a safety property produces an existential witness showing a specific violating state.

Quantifier scope matters. The formula □x. □y. R(x, y) is not the same as □y. □x. R(x, y). The first states there exists some x that relates to every y. The second states for each y there exists some x (possibly different for each y) that

relates to it. Confusing quantifier order is a common source of specification errors.

In verification contexts quantifiers appear constantly. Loop invariants often use universal quantification: all elements processed so far satisfy some property. Postconditions may use existential quantification: the function found an element with certain properties. Array and list specifications almost always quantify over indices or positions. Understanding how to read and reason about quantified statements is essential for working with formal specifications.

Sets, Relations, and Functions as Specification Tools

Sets, relations, and functions are fundamental mathematical structures that provide precise vocabulary for specifying software behavior. Even if you have not used them formally before, you have likely encountered them implicitly in programming concepts like collections, mappings, and function signatures.

A set is a collection of distinct elements with no particular order. The set {1, 2, 3} contains exactly three elements. Sets support standard operations: union (elements in either set), intersection (elements in both), difference (elements in one but not the other), subset (all elements of one set appear in another). Set notation provides concise ways to specify collections and constraints.

For example, a specification might state that the output of a filtering function is exactly the subset of input elements satisfying some predicate:

output = {x □ input | predicate(x)}

This precisely characterizes what the filter should produce without specifying how it iterates or allocates memory. The implementation can use any algorithm that produces this set.

A relation is a set of pairs describing how elements from one or more sets are connected. For example, a less-than relation on integers contains all pairs (a, b) where a < b. Relations generalize to arbitrary arity: a ternary relation might relate three elements together. In database terms relations correspond to tables, and relational algebra provides operations for querying and transforming them.

Functions are special relations where each input maps to exactly one output. The function f(x) = x + 1 maps every integer to its successor. Functions can be total (defined for all inputs in their domain) or partial (undefined for

some inputs). In verification the distinction matters: a precondition often ensures that a function is called only within its defined domain.

Function composition combines two functions: if f maps A to B and g maps B to C, then g □ f maps A to C by applying f first then g. Composition appears in verification when reasoning about pipelines of operations or sequences of transformations. Each step must produce output suitable as input to the next step.

Set theory provides notation for specifying complex properties concisely. Consider a cache data structure. Its behavior might be specified using sets:

- The cached keys are always a subset of all accessed keys
- When capacity is exceeded, evicted keys are removed from the cached set
- A lookup returns the stored value if the key is in the cached set, otherwise it misses

These specifications use set membership and subset relations precisely without getting bogged down in implementation details like which eviction policy is used or how entries are stored internally.

Boolean Algebra for Software Engineers

Boolean algebra provides systematic rules for manipulating logical expressions. Understanding these rules helps you simplify complex conditions, understand what code actually checks, and transform specifications into equivalent but more manageable forms.

The basic laws of Boolean algebra include:

Identity laws: P and true equals P. P or false equals P. Combining with the identity element leaves the value unchanged.

Domination laws: P and false equals false. P or true equals true. The dominating element determines the result regardless of P.

Idempotent laws: P and P equals P. P or P equals P. Repeating a condition does not change its meaning.

Commutative laws: P and Q equals Q and P. P or Q equals Q or P. Order of operands does not matter for AND and OR.

Associative laws: (P and Q) and R equals P and (Q and R). Similarly for OR. Grouping does not affect the result.

Distributive laws: P and (Q or R) equals (P and Q) or (P and R). P or (Q and R) equals (P or Q) and (P or R). These allow factoring and expanding expressions.

De Morgan's laws, already mentioned in the context of logical equivalences, are especially important for understanding negated conditions:

not (P and Q) = (not P) or (not Q) not (P or Q) = (not P) and (not Q)

These appear constantly when analyzing what happens when a condition fails. If a function requires both x > 0 and y < 100, the negation states that either x <= 0 or y >= 100. Understanding this helps write correct error handling and interpret counterexamples from verification tools.

Boolean algebra also clarifies short-circuit evaluation in programming languages. The expression (x != null) && (x.value > 0) relies on short-circuit semantics: if x is null the second condition is not evaluated, avoiding a null pointer dereference. Logically this is equivalent to checking both conditions, but operationally the order matters for safety. Verification tools must account for evaluation order when reasoning about code with potential side effects or errors in guard expressions.

Induction: Proving Things About Arbitrary Inputs

Induction is a proof technique for establishing that a property holds for all natural numbers (or more generally, all elements of a well-ordered structure). It is the mathematical foundation for reasoning about loops, recursion, and iterative algorithms. In verification induction appears constantly when proving loop invariants, showing recursive functions terminate, or establishing properties of data structures defined recursively.

Mathematical induction has two steps:

Base case: Prove the property holds for the smallest value, typically zero or one.

Inductive step: Assume the property holds for some arbitrary value k (this assumption is called the inductive hypothesis), then prove it also holds for k + 1.

If both steps succeed, the property holds for all natural numbers. The reasoning is that the base case establishes the property at the starting point,

and the inductive step shows that truth propagates from each value to the next. By chaining these implications infinitely you cover all values.

Consider proving that the sum of integers from 1 to n equals n(n+1)/2 for all positive integers n.

Base case: When n = 1, the sum is 1 and the formula gives 1(2)/2 = 1. They match.

Inductive step: Assume the formula holds for k: sum from 1 to k equals k(k+1)/2. We must show it holds for k + 1. The sum from 1 to k + 1 equals the sum from 1 to k plus (k + 1). By the inductive hypothesis this is k(k+1)/2 + (k+1) = (k+1)(k/2 + 1) = (k+1)(k+2)/2, which is exactly the formula for n = k + 1.

Both steps verified, so the formula holds for all positive integers n.

Structural induction generalizes this idea to recursively defined data structures. Instead of inducting on natural numbers you induct on the structure of the data. For a linked list you prove the base case for an empty list and the inductive step showing that if the property holds for the tail, it also holds when you add a new head element. This technique is essential for proving properties about trees, graphs, and other recursive structures commonly found in software.

In verification induction appears in several forms. Loop invariants are proved by induction on the number of iterations: the base case shows the invariant holds before the first iteration, and the inductive step shows that if it holds at the start of an iteration, executing the loop body preserves it. Recursive function correctness is proved by structural induction on the input. Termination proofs often use well-founded induction showing that each recursive call or loop iteration makes progress toward a base case according to some measure that cannot decrease forever.

Understanding induction is crucial because many verification arguments rely on it implicitly. When you write a loop invariant and claim it holds throughout execution, you are making an inductive argument. When you assert that a recursive function handles all inputs correctly, induction justifies the claim. Making these arguments explicit helps catch errors in reasoning about iterative and recursive code.

Proof Techniques: Direct, Contradiction, Cases

A proof is a logical argument establishing that some statement follows from given assumptions. Verification tools automate much of the mechanical work of checking proofs, but understanding proof techniques helps you construct correct specifications, debug failed proofs, and reason about program correctness when automation falls short.

Direct proof proceeds by starting from assumptions and applying logical steps to reach the conclusion. If you want to prove P □ Q, you assume P is true and derive Q through a chain of implications. Most routine verification conditions are proved directly: assuming preconditions and loop invariants hold, show that postconditions follow from program semantics.

Proof by contradiction assumes the negation of what you want to prove and shows this leads to a logical inconsistency. If assuming not P leads to a contradiction (both R and not R for some statement R), then P must be true. This technique is useful when directly proving a statement is difficult but showing its negation is impossible is easier. For example, proving that no execution reaches an error state might be done by contradiction: assume some execution does reach it, then show this violates an invariant that must hold throughout execution.

Proof by cases splits the argument into multiple scenarios and proves the desired property holds in each case separately. If you can show P holds when condition A is true and also when A is false, then P holds unconditionally. This technique appears constantly in verification because programs branch based on conditions: to prove a property after an if-then-else statement you must show it holds in both branches. Case analysis becomes more complex with nested conditionals but the principle remains the same: cover all possible cases exhaustively.

These techniques combine in practice. A typical verification argument might use case analysis on a conditional, direct proof within each branch, and contradiction to establish that certain error conditions are unreachable. Understanding which technique applies in which situation helps structure proofs effectively and communicate reasoning clearly.

Summary

This chapter covered the logical and mathematical foundations needed for verification: propositional logic with its connectives and equivalences, predicate logic with quantifiers for reasoning over collections, sets and relations as specification tools, Boolean algebra for manipulating conditions, induction for proving properties about arbitrary inputs and recursive structures, and basic proof techniques. These concepts form the language in which specifications are written and correctness arguments are expressed.

You do not need to master all of this before continuing. The concepts will appear repeatedly in subsequent chapters in concrete verification contexts, and understanding deepens through application. The key takeaway is that formal verification uses precise logical statements rather than natural language ambiguity, and the mathematical tools provide vocabulary for expressing exactly what you mean.

Chapter 3: Hoare Logic and Program Correctness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Hoare Triples: The Language of Partial Correctness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Rules of Inference for Imperative Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

P' → P, {P} C {Q}, Q → Q'

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Weakest Preconditions: Reasoning Backwards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Operational Semantics: Defining What Programs Mean

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Sequencing: ⟨C1, σ⟩ ⇓ σ', ⟨C2, σ'⟩ ⇓ σ''

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Total vs Partial Correctness and Termination

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Proof Trees for Real Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 4: Property-Based Testing as a Bridge to Formal Reasoning

From Example Tests to Universal Properties

QuickCheck, Hypothesis, and Property Frameworks

Shrinking Counterexamples: Learning from Failure

Invariants as Testable Properties

When Property Testing Succeeds and Fails

The Path from Properties to Proofs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 5: SAT Solvers and Boolean Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Satisfiability Problem and Its Ubiquity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Conjunctive Normal Form: Encoding Problems as Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

How SAT Solvers Work: DPLL and CDCL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Using SAT Solvers in Practice: MiniSat, Z3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Encoding Real Constraints for SAT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Limits of Pure Boolean Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 6: SMT Solvers and Constraint Solving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

From SAT to SMT: Adding Theories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Core Theories: Arithmetic, Arrays, Bitvectors, Strings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Z3 and Other Major SMT Solvers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Encoding Programs as Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Solver-Aided Programming Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Interpreting Sat/Unsat and Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 7: Symbolic Execution and Path Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Symbolic Values and Path Conditions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Exploring Branches Systematically

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Concolic Execution: Mixing Concrete and Symbolic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Bounded Model Checking via Symbolic Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Tools: KLEE, angr, CBMC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Path Explosion and Practical Mitigations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 8: Abstract Interpretation and Static Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Idea of Abstraction: Precision vs Soundness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Lattices and Fixpoints Made Intuitive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Common Analyses: Nullability, Ranges, Taint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Frama-C and the WP Plugin

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

SPARK/Ada for Safety-Critical Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Trade-offs Between Soundness and Scalability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 9: Model Checking and Temporal Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

State Spaces and Transition Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Linear-Time vs Branching-Time Temporal Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

LTL and CTL: Specifying Behavioral Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

How Model Checkers Explore State Spaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

TLA+ for Distributed Systems Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

State Explosion and Symmetry Breaking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 10: Deductive Verification with Contracts and Annotations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Deductive Verification Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Dafny: Syntax, Contracts, and Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Loop Invariants and Recursive Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Framing Conditions and Modularity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Verus and Rust Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Debugging Failed Proofs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 11: Separation Logic and Memory Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Problem of Shared Mutable State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Separating Conjunctions and Spatial Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Points-to Assertions and Heap Predicates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Verifying Pointer Manipulation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Prusti, Creusot, and Rust Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Concurrent Separation Logic Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 12: Concurrent Program Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

What Goes Wrong in Concurrent Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Linearizability and Atomicity Specifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Race Detection vs Race Proofs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Lock-Based Reasoning and Ownership

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Verifying Concurrent Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Tools: IronFleet, F*, Viper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 13: Distributed Systems and Protocol Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Unique Challenges of Distributed Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Specifying Consensus and Replication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Alloy for Lightweight Structural Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Verifying Paxos, Raft, and CRDTs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Handling Timeouts, Failures, and Network Partitions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

IronFleet: Verified End-to-End Distributed Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 14: Theorem Proving with Proof Assistants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

When You Need a Proof Assistant

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Dependent Types and Propositions-as-Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Interactive Proof Development in Lean

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Tactics, Automation, and Proof Terms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Coq/Rocq for Certified Software

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Isabelle/HOL for Large-Scale Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 15: Verified Compilers and Systems Software

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Why Verify Compilers? The CompCert Story

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Verified Operating System Kernels (seL4, Fuchsia)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Cryptographic Implementations in F* and Others

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Proof-Carrying Code and Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Smart Contract Verification on Blockchains

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Lessons from Million-Line Verified Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Chapter 16: Engineering Practice — Adoption, Workflows, and Culture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Choosing What to Verify: Risk vs Effort

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Trusted Computing Base and Its Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Integrating Verification into CI Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Specification Review and Proof Maintenance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Combining Verification with Testing and Fuzzing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Introducing Formal Methods to Engineering Teams

Summary

Major Case Study: From Tests to Verified System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Component: Requirements and Initial Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Baseline Testing and Its Blind Spots

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Adding Property-Based Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

SMT-Based Contract Checking with Dafny

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Model Checking the State Machine in TLA+

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Machine-Checked Proofs for Critical Properties

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

CI Integration and Regression Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

What We Learned: Costs, Benefits, Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Conclusion: The Future of Verified Software Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Where Formal Verification Has Succeeded

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

AI and Language Models in Verification Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

The Remaining Gaps Between Research and Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

A Pragmatic Philosophy for Verified Software

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

Getting Started Tomorrow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/formalverificationforworkingsoftwareengineers.