Rigorous Digital Engineering

High-Assurance Cyber-Physical Systems

What is Rigorous Digital Engineering?

Rigorous Digital Engineering (RDE) is the integration of:

- **Formal Methods**: Mathematically grounded reasoning
- **Digital Modeling**: Executable, analyzable models (twins/threads)
- **Engineering Discipline**: Structured methodologies across system layers

It applies to **software, hardware, firmware, embedded systems, safety-critical systems**, **CPS**, and more.

Rigorous

- Uses **formal logic**, **mathematical proof**, and **types**
- Emphasizes **Design-by-Contract** and **Correct-by-Construction**

Digital

- Models are **mechanized**, **denotational**, **computational**, and **executable**
- Enables digital twins (model $\leftrightarrow$ reality) and assurance chains (trace links)

Engineering

- Applies across disciplines: domain, requirements, system, software, hardware, safety, security
- Uses well-defined **processes**, **toolchains**, and **evidence-driven** assurance

Rigorous Digital Engineering

Rigorous

- Uses **formal logic**, **mathematical proof**, and **types**
- Emphasizes **Design-by-Contract** and **Correct-by-Construction**

What Is Design by Contract?

Design by Contract (DbC) is a programming methodology where software components define and enforce:

- **Preconditions**: What must be true before execution
- **Postconditions**: What will be true after execution
- **Invariants**: What must always remain true about system state

Each component behaves like a contract party:
it guarantees correct behavior — if the preconditions are satisfied.

Example: withdraw(amount)

A bank account method:

Condition	**Requirement**
Precondition	`amount > 0 and amount <= balance`
Postcondition	`balance = old_balance - amount`
Invariant	`balance >= 0`

If the caller provides a valid amount, the function guarantees a valid result, and the balance never goes negative.

Why Use Design by Contract?

- Defines clear expectations between caller and callee
- Prevents misuse and encourages defensive programming
- Helps identify bugs precisely: who broke the contract?
- Enables automated checking, testing, and formal verification

What Is Correct-by-Construction?

Correct-by-Construction (CbC) is a methodology where systems are designed and implemented in such a way that correctness is guaranteed by the process itself.

- Errors are prevented, not just detected
- Verification is integrated into development
- Each refinement step is provably safe and justified

CbC Key Principles

- Specification-driven development
- Stepwise refinement from models to implementation
- Formal verification integrated with design
- Automation using synthesis, type systems, and contracts

CbC Benefits for Engineers

- Reduces post-implementation defects
- Ensures early alignment with requirements
- Lowers cost of testing and certification
- Produces traceable, verifiable artifacts

CbC Examples in Practice

- Synthesizing hardware modules from verified models
- Generating safety-critical code from Simulink or SCADE
- Using TLA+, F*, Coq, or Haskell with contracts and proofs
- Embedding formal logic in DSLs and generators

CbC in the Workflow

1. Model system behavior and interfaces
2. Refine design through provable, incremental steps
3. Generate implementation code from verified artifacts
4. Maintain traceability from model to code

Rigorous Digital Engineering

Digital

- Models are **mechanized**, **denotational**, **computational**, and **executable**
- Enables digital twins (model $\leftrightarrow$ reality) and assurance chains (trace links)

Models That Are More Than Diagrams

In rigorous digital engineering, models aren't just sketches.

They are:

- Mechanized
- Denotational
- Computational
- Executable

Mechanized Models

Mechanized means the model is written in a way that software tools can read and analyze it.

- Not just human-readable — it's machine-processable
- Enables static checking, proof generation, code synthesis
- Examples:
  - Proof generation: Coq, Isabelle/HOL, Lean
  - Code generation: OCaml, Haskell, Simulink, SCADE

Tools understand and work with the model directly.

Denotational Models

Denotational means the model has a precise mathematical meaning.

- Maps to formal semantics
- You know exactly what every part of the model *means*
- Enables formal reasoning and guarantees

Example: a TLA+ model of a protocol defines its logic in set theory and temporal logic.

Computational Models

Computational means the model can be used to simulate behavior.

- The model describes how outputs evolve based on inputs
- Can be run through simulators
- Useful for early validation and test generation

Example: a Simulink control model that responds to real-time signals.

Executable Models

Executable means the model can be run like a program.

- It defines actual behavior, not just structure
- Can generate or become running software
- Bridges modeling and implementation

Examples:

- State machines that can be stepped through
- Domain-specific languages that compile into real code

Modeling Summary

Quality	**What It Means**
Mechanized	Tool-readable and analyzable
Denotational	Mathematically defined and unambiguous
Computational	Supports simulation and behavioral analysis
Executable	Can run or generate running software

These qualities make formal models practical, verifiable, and useful in real systems.

What Are Assurance Chains?

Digital threads are **traceable**, connected chains of information that link **artifacts** across the system lifecycle.

They enable structured, end-to-end traceability from:

- Requirements
- Design and modeling
- Code and tests
- Deployment and runtime data

Assurance Chains = Trace Links

A assurance chain includes **typed, semantic links** between artifacts:

- "Implements..."
- "Refines..."
- "Validates..."
- "Depends on..."

This makes it possible to follow the path from **high-level requirements** to **low-level implementation** and back.

What Assurance Chains Enable

- Change impact analysis
- Evidence for assurance and certification
- Consistency across lifecycle stages
- Model-code-test alignment
- Compliance with safety and security standards

Practical Use Cases

- **Aerospace**: Track how a flight safety requirement is implemented and tested
- **Medical**: Show compliance from spec to firmware in a device
- **Software**: Know which tests must be rerun when a model changes

Digital threads bring **visibility**, **structure**, and **accountability** to complex systems.

Assurance Chain Summary

Digital threads support:

Feature	**Benefit**
Full traceability	Fewer surprises during audits
Lifecycle linking	Synchronized engineering artifacts
Semantic meaning	Know what a connection represents
Assurance support	Easier evidence gathering and reuse

They make complex, critical systems **navigable**, **explainable**, and **verifiable**.

Rigorous Digital Engineering

Engineering

- Applies across disciplines: domain, requirements, system, software, hardware, safety, security
- Uses well-defined **processes**, **toolchains**, and **evidence-driven** assurance

RDE Spans the Entire Engineering Stack

Rigorous Digital Engineering (RDE) applies across multiple disciplines:

- Domain Engineering
- Requirements Engineering
- System Engineering
- Software Engineering
- Hardware Engineering
- Safety Engineering
- Security Engineering

It's not just about code or models — it's a holistic approach to engineering.

Domain Through Security: What It Means

Domain Engineering
- Models real-world constraints, physics, environments

Requirements Engineering
- Structured or formal requirements
- Traceable to code and tests

System Engineering
- Architecture and interactions
- Verified component integration

Software Engineering
- Verified contracts, static analysis, proofs

Hardware Engineering
- Formal verification of cyber-physical systems

Safety Engineering
- Hazard analysis, fault tolerance, safety claims

Security Engineering
- Cryptographic proofs, secure protocol guarantees

Hardware Modeling: Beyond Digital Logic

In many formal engineering discussions, "hardware models" often focus on:

- RTL (register-transfer level) logic
- FPGAs and ASICs
- Bit-accurate digital behavior

But in **controls and cyber-physical systems**, hardware modeling must go further.

It must include the **physical devices** being controlled — actuators, sensors, and the environment itself.

Hardware modeling — Why Physical Hardware Matters

The physical world **defines the meaning** of signals.

- Sensor data depends on device dynamics and calibration
- Actuator commands must respect timing, limits, and failure modes
- Control logic relies on accurate assumptions about physical response

If the model ignores these aspects, it may be formally correct, but **physically invalid** or unsafe.

Hardware modeling — Toward Multi-Layer Hardware Modeling

To ensure end-to-end system correctness, models must span:

- **Digital hardware**: logic, registers, memory maps
- **Embedded interface**: how signals are read and written
- **Device behavior**: dynamics, range, latency, noise
- **Physical environment**: physics, constraints, external disturbances

Only by modeling these layers can we build control systems that are **verifiably correct** in the real world.

RDE Uses Well-Defined Processes

RDE promotes structured, auditable workflows:

- Model $\rightarrow$ Verify $\rightarrow$ Trace $\rightarrow$ Deliver
- Each step supports automated or formal reasoning
- Aligns with safety and security standards

Examples:

- DO-178C (avionics)
- ISO 26262 (automotive)
- IEC 61508 (industrial safety)

Toolchains in RDE

RDE depends on reliable, connected tools:

- **Modeling**: SysML, AADL, Simulink
- **Formal Verification**: Coq, Frama-C, TLA+
- **Simulation & Synthesis**: SCADE, Simulink, SystemVerilog
- **Analysis**: SAT/SMT solvers, static checkers

These tools produce outputs that connect — through shared artifacts or trace links.

Evidence-Driven Assurance

RDE focuses on **evidence**, not just confidence.

- Formal proofs (e.g., safety properties, functional correctness)
- Test results and coverage
- Verified traceability between
  - Requirements
  - Design
  - Code
  - Tests
- This evidence supports
  - Certification
  - Regulatory review
  - Maintenance and change management

Why Should Engineers Care?

- Bugs and flaws in critical systems can cost millions
- Traditional V&V (testing, inspection) catches only a fraction of issues
- Regulatory burden is growing: **evidence matters more than code**
- RDE lets you:
  - Catch design flaws before implementation
  - Generate analyzable artifacts directly from models
  - Provide **traceable, machine-checkable evidence of correctness**

What Does RDE Replace or Improve?

Instead of:

- Manual documentation
- Boilerplate code
- After-the-fact testing
- Informal specifications

Use:

- Executable models
- Code synthesis
- Formal verification
- Traceable requirements

Practical Benefits

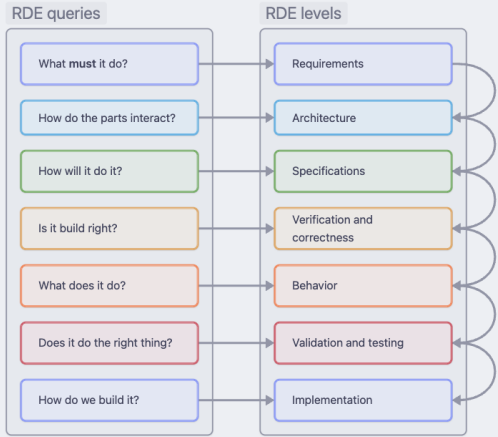
- **Fewer bugs**, earlier in development
- **Reduced rework**, thanks to better specs
- **Higher assurance** with less manual effort
- **Faster reviews** with machine-checkable evidence
- **Better integration** between software and hardware development

Technical Foundations

- **Formal Methods**: The foundation of RDE
  - Hoare logic, SAT/SMT solving, model checking, theorem proving
- **Executable Semantics**: Models aren't just documentation—they run!
- **Refinement**: Relationships between abstraction layers are provable
- **Traceability**: Every element links to its specification or evidence

RDE in System Development Lifecycle

1. **Requirements**: Structured, traceable, ideally semi-formal
2. **Design**: Architecture modeling (AADL, SysML)
3. **Implementation**: Contract-based programming or synthesis
4. **Verification**: Formal verification, runtime monitoring, test generation
5. **Deployment**: Use traceability to validate final system
6. **Maintenance**: Models evolve with implementation (not as afterthought)


RDE queries
What must it do?
How do the parts interact?
How will it do it?
Is it build right?
What does it do?
Does it do the right thing?
How do we build it?
RDE levels
Requirements
Architecture
Specifications
Verification and correctness
Behavior
Validation and testing
Implementation

Example Tools and Technologies

Modeling & Specification

- SysML, AADL, UML
- ACSL, JML, F*, TLA+

Programming & Verification

- SPARK Ada, MISRA C, Rust
- Frama-C, Coq, Isabelle, Lean

Simulation & Synthesis

- Simulink, SCADE
- SystemVerilog Assertions

IDEs & Frameworks

- Eclipse + OSATE, VS Code, mbeddr, OpenMBEE
- Docker/Nix for reproducible toolchains

Toolchains You (May) Already Know

Familiar Tool	**RDE-Compatible Enhancement**
C/C++	Frama-C, ACSL annotations
MATLAB	Simulink + formal checks
Verilog	SystemVerilog + assertions
UML/SysML	Add formal semantics + traceability
Excel for requirements	FRET or SpeAR for structured reqs

Real-World Engineering Challenges

- Engineers don't want to stop coding to "draw diagrams"
- Many modeling tools are clunky or proprietary
- Legacy code has no model—it's hard to retrofit
- Formal tools require training and mindset shift

Fact vs. Myth

Myths

- "Formal methods are only for academics"
- "It takes too long and costs too much"
- "You need a PhD to use these tools"

Facts

- They're actively used in aerospace, defense, automotive, and nuclear
- Proper use reduces cost and risk, especially in regulated industries
- Many tools integrate with familiar environments (e.g., VS Code, Git)

Approaches: How to Get Started

- Start small: instrument existing systems with contracts or assertions
- Use **modeling** to document **architecture**, even if not executable at first
- Use **formal tools at key interfaces** (e.g., cryptographic components, fault logic)
- Ensure toolchains produce evidence that can be reused in certification

Summary: Why RDE Matters

- Engineers can build better systems, faster
- Assurance can be integrated — not bolted on
- Formal evidence is the future of compliance and safety
- RDE is a practical, repeatable, and teachable methodology