Formal Methods for Safety-critical software

Introduction to Formal Methods

Safety-critical systems

- Software is now a critical component of most safety-critical systems
- Software by itself can’t do harm, but systems it integrates can
- Consequences of failures:
  - Injury or loss of life
  - Damage to the environment
  - Loss of equipment
  - Loss of data
  - Loss or reputation
  - …
- What classifies as safety-critical?

Avionics

Railway

Banking

Spacecraft

Driver-assistance

Medical devices

Image sources: wikipedia

Example: Therac-25 (1989)

- Medical radiation machine
- Radiation overdoses, injuries and deaths
- Software bugs weren’t even the critical: poor requirement specification and design
- Moved safety control from hardware to software but not properly verified

KILLED BY A MACHINE: THE THERAC-25

by: Adam Fabio

145 Comments

October 26, 2015

The Therac-25 was not a device anyone was happy to see. It was a radiation therapy machine. In layman’s terms it was a “cancer zapper”; a linear accelerator with a human as its target. Using X-rays or a beam of electrons, radiation therapy machines kill cancerous tissue, even deep inside the body. These room-sized medical devices would always cause some collateral damage to healthy tissue around the tumors. As with chemotherapy, the hope is that the net effect heals the patient more than it harms them. For six unfortunate patients in 1986 and 1987, the Therac-25 did the unthinkable: it exposed them to massive overdoses of radiation, killing four and leaving two others with lifelong injuries. During the investigation, it was

Example: Ariane 5 (1996)

- Launch vehicle developed by ESA
- Crashed in 40 seconds, millions $ cost
- A software bug: converting a 64-bit value in smaller precision (in a module that was not even needed)
- Error not caught and handled correctly

A space error: $370 million for an integer overflow

02.09.2016 / HOWNOT2CODE

Start. 37 seconds of flight. KaBOOM! 10 years and 7 billion dollars are turning into dust.

The programmers were to blame for everything.

Four satellites, 2,600 lb, of the Cluster scientific program (study of the solar radiation and Earth's magnetic field interaction) and a heavy-lift launch vehicle Ariane 5 turned into "confetti" June 4, 1996.

Example: Denver Airport Baggage Handling (1995)

- Delayed by 16 months, hundreds millions $ in losses, eventually cancelled
- Poor design, solution was never feasible, ad hoc solution

Almost all grave software problems can be traced to conceptual mistakes made before programming started

Failed automated baggage system at Denver International Airport.

An architectural marvel when it opened 11 years ago, the new Denver International Airport's high-tech jewel was to be its automated baggage handler. It would autonomously route luggage around 26 miles of conveyors for rapid, seamless delivery to planes and passengers. But software problems dogged the system, delaying the airport's opening by 16 months and adding hundreds of millions of dollars in cost overruns. Despite years of tweaking, it never ran reliably. Last summer airport managers finally pulled the plug—reverting to traditional manually loaded baggage carts and tugs with human drivers. The mechanized handler's designer, BAE Automated Systems, was liquidated, and United Airlines, its principal user, slipped into bankruptcy, in part because of the mess.

The high price of poor software design is paid daily by millions of frustrated users. Other notorious cases include costly debacles at the U.S. Internal Rev-

project); the Federal Bureau of Investigation (a $170-million virtual case-file management system was scrapped in 2005); and the Federal Aviation Administration (a lingering and still unsuccessful attempt to renovate its aging air-traffic control system).

Such massive failures occur because crucial design flaws are discovered too late. Only after programmers began building the code—the instructions a computer uses to execute a program—do they discover the inadequacy of their designs. Sometimes a fatal inconsistency or omission is at fault, but more often the overall design is vague and poorly thought out. As the code grows with the addition of piecemeal fixes, a detailed design structure indeed emerges—but it is a design full of special cases and loopholes, without coherent principles. As in a building, when the software's foundation is unsound, the resulting structure is unstable.

Managers involved in high-profile

would be right. Developers rar
ticulate their designs precisely an
lyze them to check that they embo
desired properties. But with com
now flying airplanes, driving trai
cars, and running most of the fin
communications, trading and
duction machinery of the world,
has an urgent need to improve so
dependability.

Now a new generation of so
design tools is emerging [*see box o*
74]. Their analysis engines are sim
principle to tools that engineers ir
ingly use to check computer har
designs. A developer models a so
design using a high-level (sum
coding notation and then applies
that explores billions of possible
tions of the system, looking for u
conditions that would cause it to
in an unexpected way. This p
catches subtle flaws in the design
it is even coded, but more import
results in a design that is precise

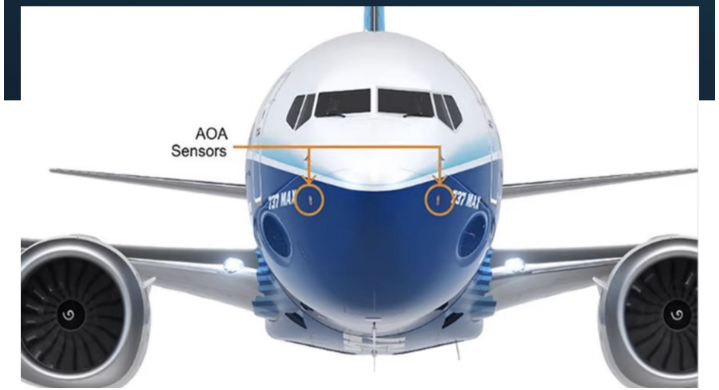
Example: Boeing 737 MAX (2019)

- Two crashes, hundreds of deaths
- Software safety controller to fix an intractable hardware problem
- Poorly designed: relied on single sensor, no redundancy

Killer software: 4 lessons from the deadly 737 MAX crashes

By **Matt Hamblen** • Mar 2, 2020 06:23pm

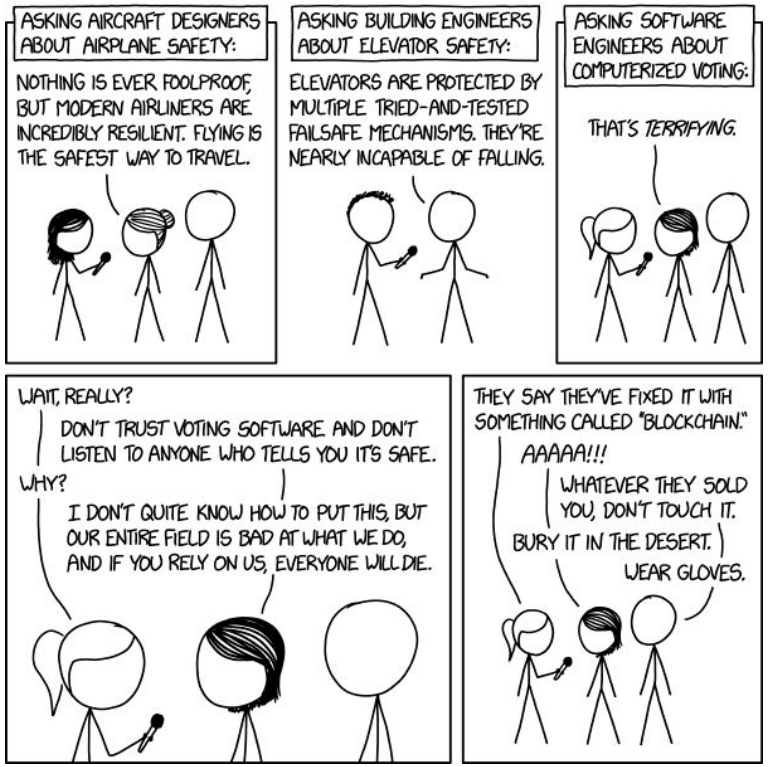
Acceleration/Vibration | Aerospace/Military | Sensor Applications | Software



Boeing 737 MAX planes are quipped with two angle of attack (AOA) sensors on either side of the fusilage nose, but a flight control software fix called MCAS was relying on data from just one of the sensors in the Lion Air crash in 2018, authorities said. (Boeing)

It's been widely reported that Boeing's decision to use a flight control software fix known as MCAS in its 737 MAX planes was one of the key factors that led to two crashes that killed 346 people

Where’s the engineering on software engineering?



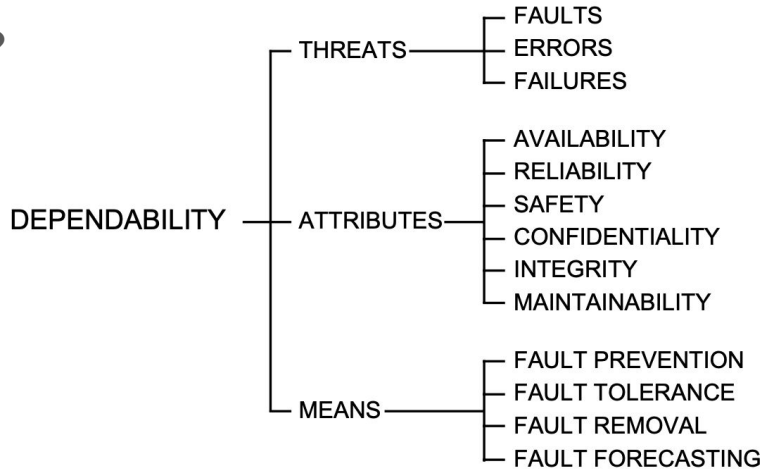
Source: https://xkcd.com/

Dependable Systems

- We want our critical systems to be **safe**, **secure**, **available**, …
- Software is only a sub-system (although an increasingly relevant one)
- But without dependable software, there are no dependable systems

How can we build dependable software?

- Identify and classify threats
- Specify desirable attributes
- Define means to guarantee them



Source: Avizienis et al.

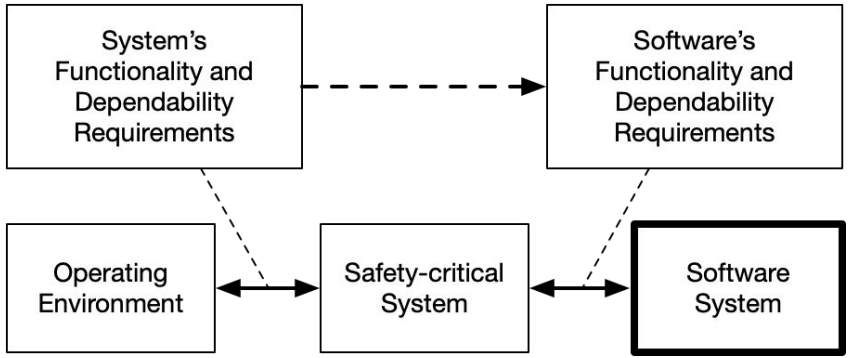
Dependability

The dependability of a system is the ability to avoid failures that are more frequent and more severe than is acceptable.

- We cannot protect our systems about all kinds of failure
- We cannot guarantee complete absence of failures
- We need to specify the **dependability requirements** of our systems

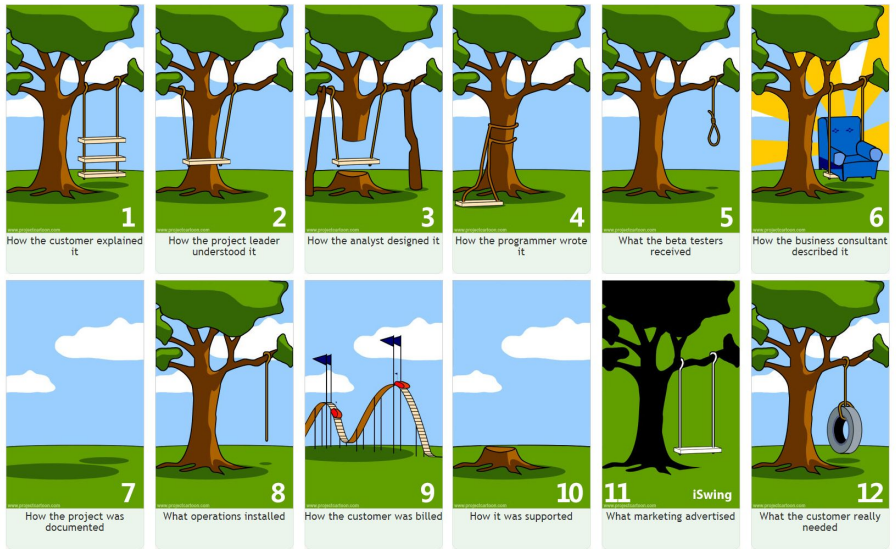
Dependability Requirements

- Transversal to the complete system
- Are orthogonal of the functional requirements, must be reasoned about independently
- Software system inherits the requirements from its interface with the environment
    - It’s a contract with the rest of the system



Requirements vs Specifications

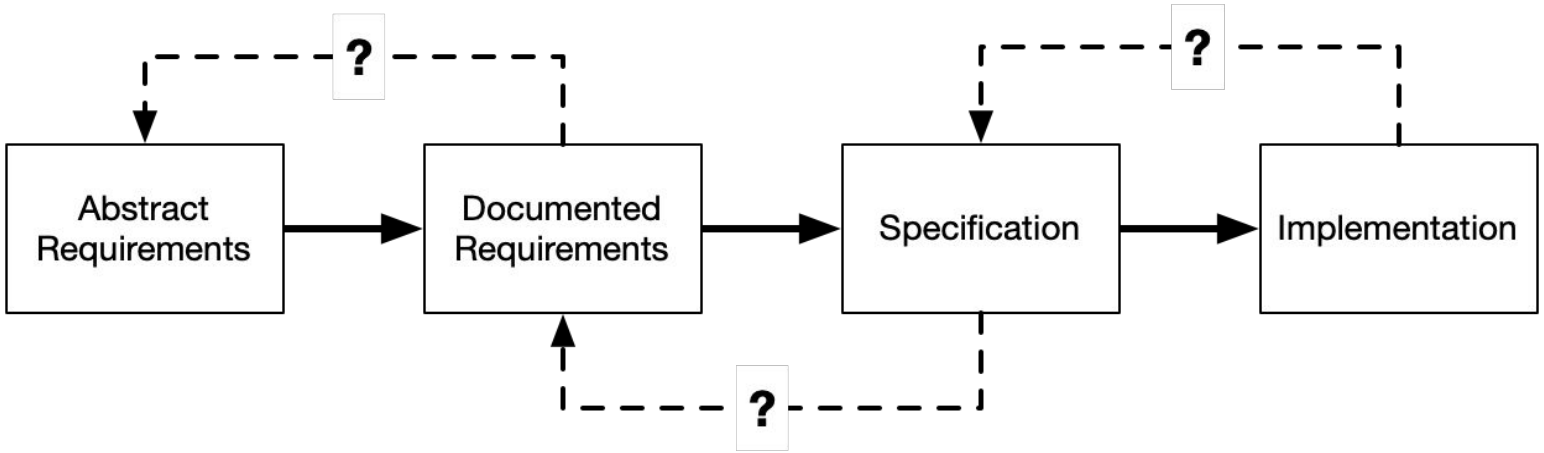
- Requirements are set by the customer
- Specifications are derived by the developing team: what the software should do to meet the requirements
- Guaranteeing that the specification reflects the requirements is hard
- Even worse:
  - Documented requirements may not reflect the “abstract” requirements
  - Same may apply to specifications…



Source: http://www.projectcartoon.com/cartoon/1111

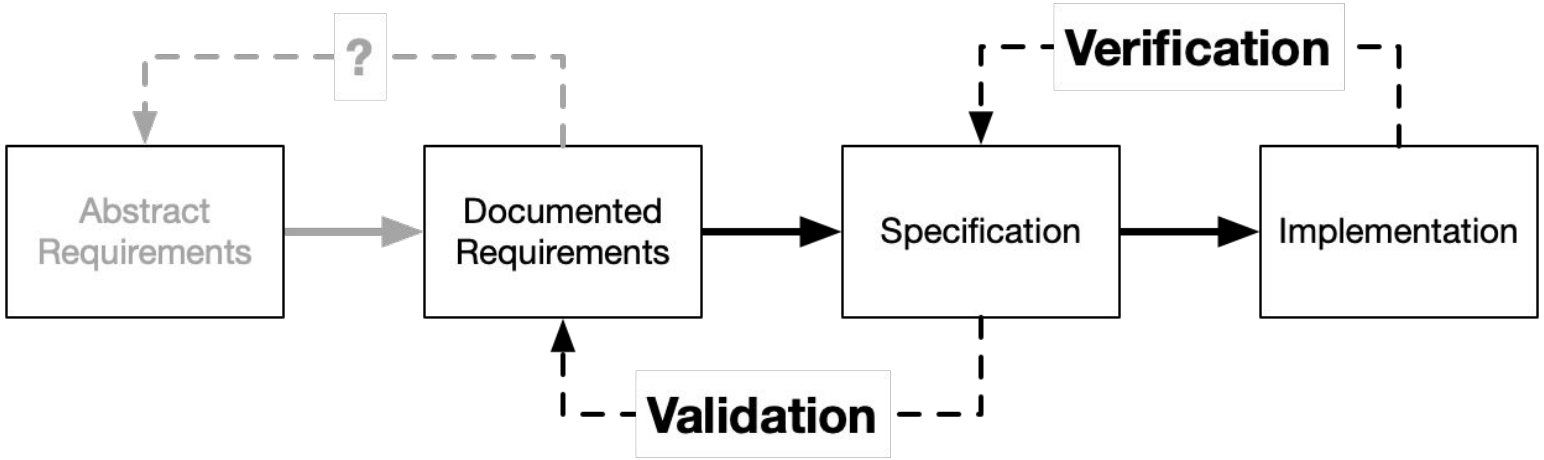
Why do systems fail?

- Documented requirements do not reflect abstract requirements
- Specification of the system does not reflect the documented requirements
- Implementation of the system is not a refinement of the specification



Why do systems fail?

- Documented requirements do not reflect abstract requirements
- Specification of the system does not reflect the documented requirements
- Implementation of the system is not a refinement of the specification



Dependability Attributes

Dependability requirements are defined according to:

- **Reliability:** continuity of correct service
- **Availability:** readiness for correct service
- **Safety:** absence of catastrophic consequences
- **Integrity:** absence of unauthorized disclosure of information
- **Confidentiality:** absence of improper system state alterations
- **Maintainability:** ability to undergo repairs and modifications

Dependability Attributes: Reliability and Availability

Formally, **reliability** is the probability the system will operate correctly during a period of time

- This period is key: airplane hours, spacecraft years

Formally, **availability** is the probability the system will be operational at each particular instant

- Systems may fail, but they must repair within reasonable time

Dependability Attributes: Safety

Safety is difficult to formally quantify:

- What is the acceptable rate of a failure if human injury is involved? Or environmental damage?
- In safety-critical system, safety is seen as an absolute
  - Particularly complex task, must predict and handle all possible failures
  - In software components, the main target of *formal methods*
- If some failures are expected, safety measured as some quantified “loss” over time

Dependability Attributes: Integrity and Confidentiality

Confidentiality and **Integrity** are mostly related to security concerns.

- As with safety, it is often seen as an absolute
- Needs carefully analysis of consequences, the attackers, the permissions…
- Also a typical target of formal methods

Dependability Attributes: Maintainability

Maintainability is also difficult to quantify

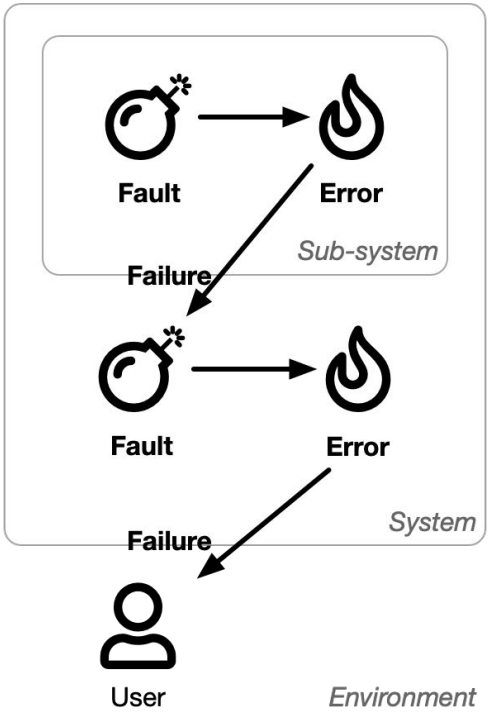
- How to define the ability to modify a system?
- Software in particular evolves over time
  - And many faults are introduced during upgrades
- Often relies on code metrics and standards to keep code simple and readable

Failure

- Dependability is defined around the notion of **failure**
- A failure occurs when
  - The system is in an *erroneous* state (an error occurred)
  - That state is visible to the *external* environment
- (Note: here an error is an event, not a “bug”)
- To quantify failures, we must identify and handle the causes of errors

Faults

- A **fault** is the cause of an erroneous state
  - Degradation fault (hardware, more predictable)
  - Design fault (all software faults, both in specification and implementation)
- We need to predict “all” possible faults of our system
  - We cannot deal with unanticipated faults!
- How to deal with predicted faults?
  - Avoidance
  - Elimitation
  - Tolerance
  - Forecasting



Fault Avoidance

- Simply, do not introduce faults in the system
- Should always be the first strategy, but never fully effective
- Particularly important for software due the complexity
- In hardware, degradation faults can never be avoided

Fault Elimination

- Faults were introduced, but are detected and removed before deployment
- In software, testing is commonly used
- But testing can never have full coverage
- Safety-critical systems need more advanced techniques

Fault Tolerance

- Developers may choose to leave some faults (or be unable to fix them)
- In those cases, faults must be *tolerated* (e.g., through redundancy)
- System continues to provide acceptable behaviour even after failures

Fault Forecast

- If faults cannot be avoided, eliminated or tolerated, the developer may leave faults if failures are acceptable
- The consequences of failure must be fully studied, and a statistical model of their occurrences developed
- This is particularly difficult for software: design faults are difficult to model (in contrast to degradation faults)

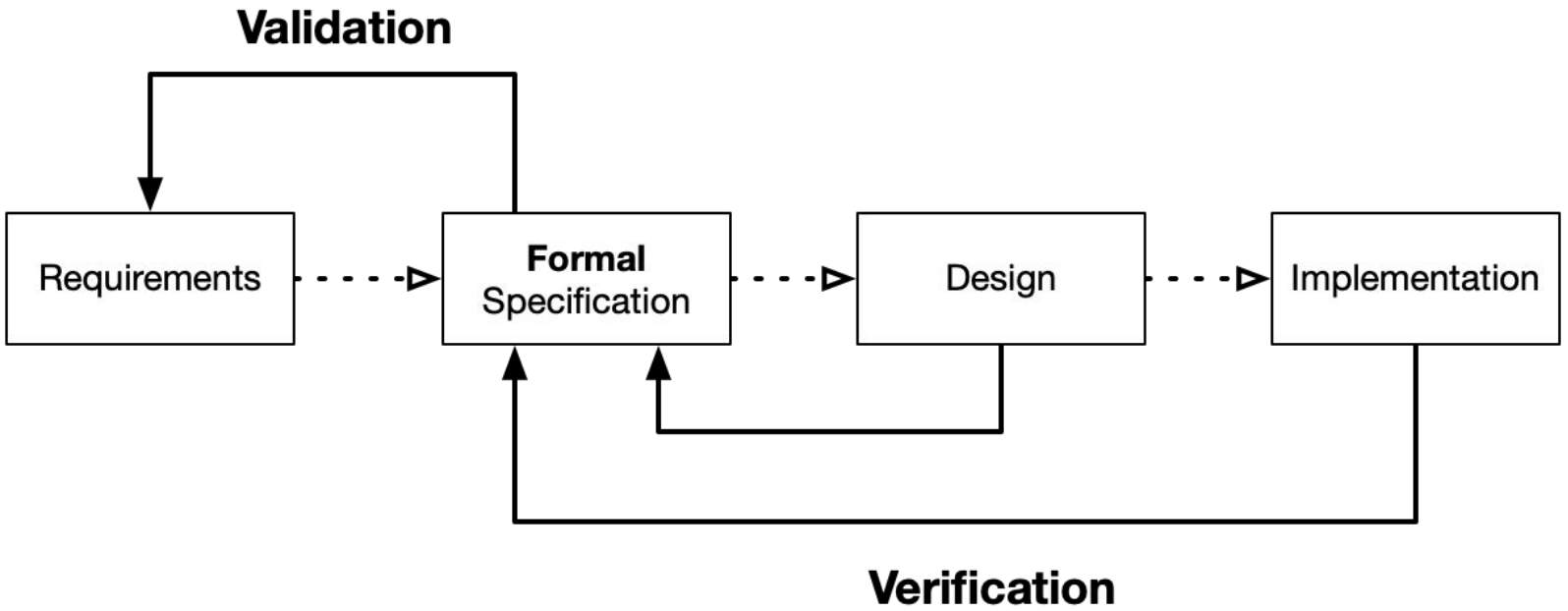
Software Dependability

- Dependability is a concern in any engineering field, but **software** has its own particularities
- Software is much more complex than other systems (that's why it's being adopted in every area)
- Faults are logical, and more unpredictable (unlike degradation faults), so we should focus on fault avoidance and elimination
- Enter *formal methods*: techniques for the development of dependable software

Formal Methods

- Provide a mathematical foundation to analyze software systems
- Allow the **validation** and **verification** tasks needed for dependability
  - Are we building the *right thing*?
  - Are we building the *thing right*?
- Common practice in other engineering fields:
  - Testing is only used to confirm the analysis, not to establish conclusions
- Software is much harder to formalize (it’s behaviour is not continuous)

Formal Methods in the Development Cycle



- **Specifications** are a central artefact in the process

Formal Specifications

- Faults in the specification are the most expensive to eliminate: requires adapting design and implementation
- In formal methods, specifications must be formally defined
  - Unambiguous, precise, formal semantics
  - Contrast with UML
- Formal specification *languages* provide a way for users formalize requirements
- They are backed by tools that support validation and verification
- Examples:
  - ACSL, JML, **Dafny**, OCL, VDM, Z, TLA+, **Alloy**, B-Method, …

Techniques Behind Formal Methods

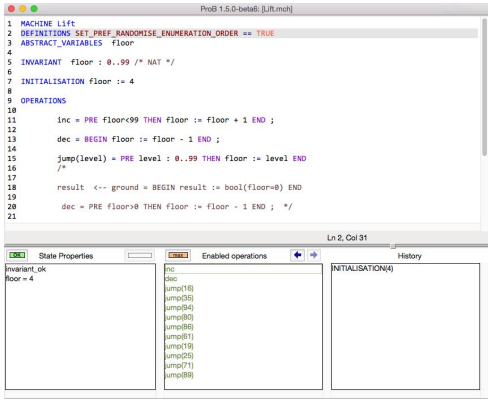
- Two main approaches to check whether a property holds
  - **Model checking:** exhaustively explore all possible states
  - **Automated/Interactive theorem proving:** prove statement deductively
- These can be used in different stages of the development process
- Often used in the backend, users do not interact directly

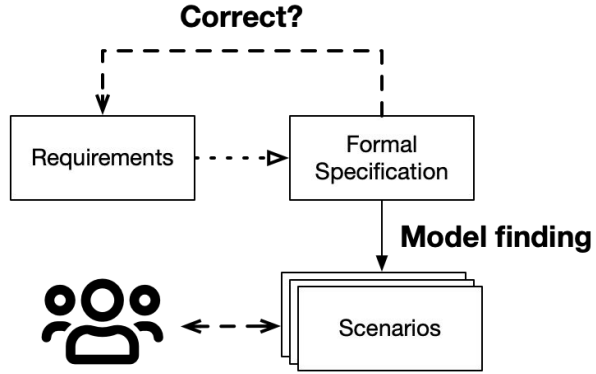
Techniques Behind Formal Methods

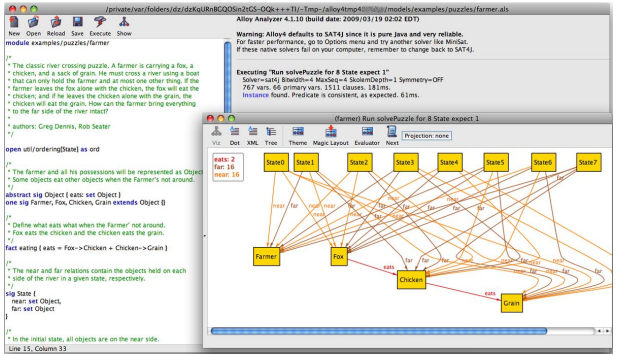
- The problems we want tackle are usually **undecidable**
- Must sacrifice either
  - Completeness (limit search space, e.g., model checkers)
  - Expressiveness (support restricted logics, e.g., automated theorem provers)
  - Automation (needs user input, e.g., interactive theorem provers)

FMs for Validating Specifications

- *Model finding* to animate specifications and perform scenario exploration
- Scenarios guaranteed to obey the specification, validated with stakeholders
- Use model checkers underneath

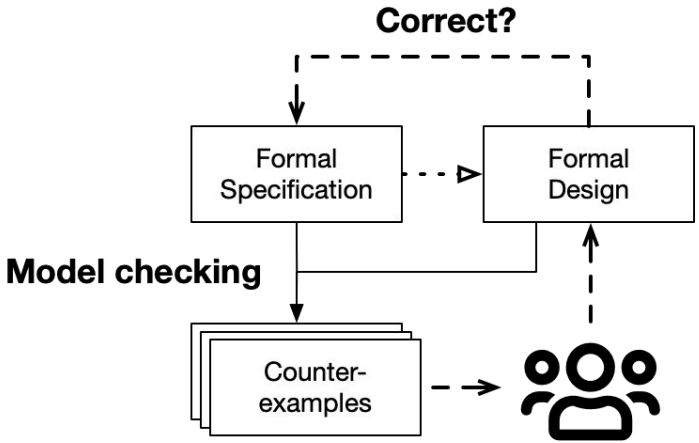






FMs for Verifying Designs

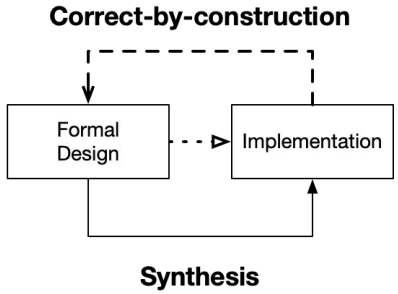
- Formalize the design, use *model checking* to verify it against the specification
- Typically, the model is state machine
- High-level languages provided
- Specification in some of *temporal logic*
  - Something bad will never happen
  - Something good will eventually happen



FMs for Correct-by-Constructions Implementations

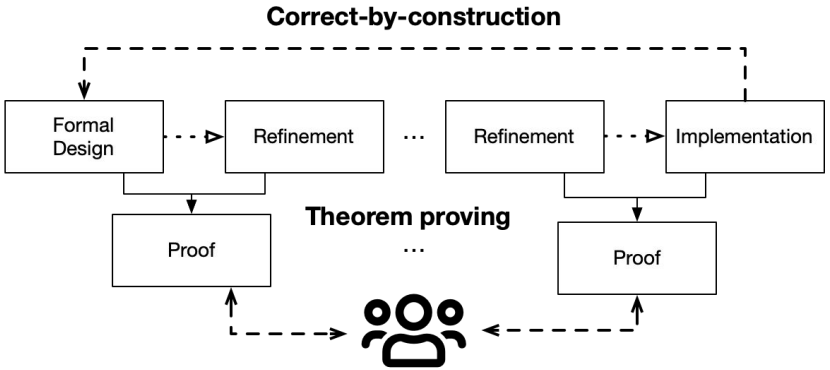
- Formalize the design, use *program synthesis* to obtain implementation
- Difficult for general-purpose programs

MATLAB® & SIMULINK®



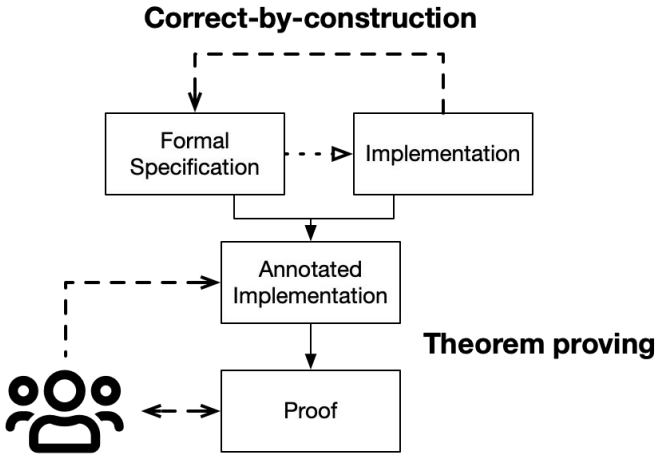
- Formalize the design, use *refinement steps* until actual implementation
- Use theorem provers to show that each step preserves behaviour

ATELIER B



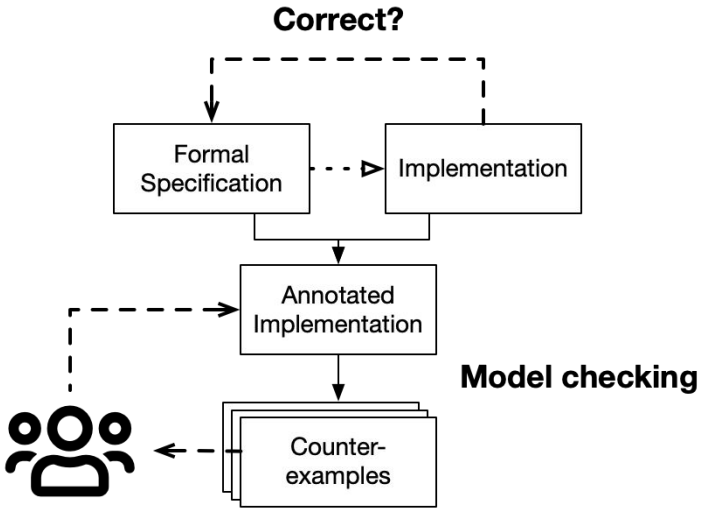
FMs for Correct-by-Constructions Implementations

- Formalize the specification as contracts, use *deductive reasoning* to check them
- Part of the compilation process (incorrect programs don't execute)
- Uses theorem provers: may be interactive or require additional inputs



FMs for Implementation Verification

- Formalize the specification as *assertions*, use *model checking* to verify program
- A model of the program is extracted from code
- Automatic, but bounded search space means limited loop unrolls

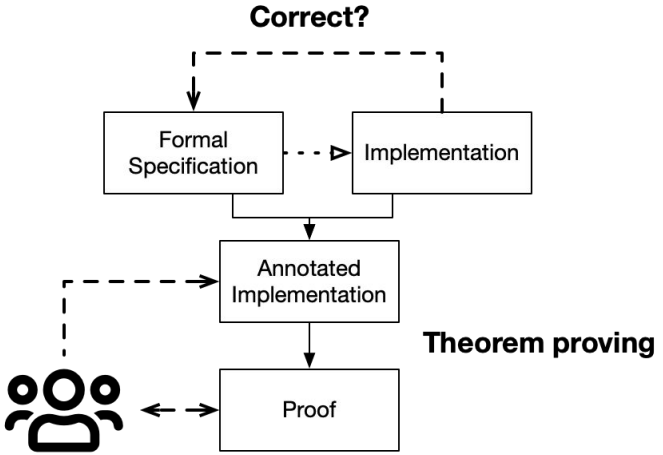


FMs for Implementation Verification

- Formalize the specification as *assertions*, use *deductive reasoning* to verify program
- Proof conditions are extracted from code
- Uses theorem provers: may be interactive or require additional inputs

frama C
Software Analyzers

VeriFast

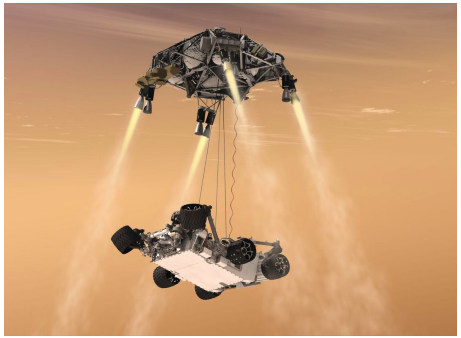


Dependability Assessment

- Quantitative Assessment
  - Create a probabilistic model of the system, assess dependability attributes
  - Very difficult for software, and doesn’t apply to absolute requirements
- Prescriptive Standards
  - Control the development process (lifecycle, documentation, style, ...)
  - Cannot guarantee actual correctness but give structure and promote best practices
- Rigorous Arguments
  - Built a *dependability case*, breaking the argument into smaller pieces and providing evidences
  - Evidences can be adapted to each component

Formal Methods in the Wild

Charles de Gaulle shuttle, ClearSy



Curiosity Rover, NASA JPL

Rotterdam storm surge barrier, LogicaCMG

iFACTS Air Traffic Control, Altran Praxis

contributed articles

DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

S3 is just c
vices that stor
customers hav
safeguard that
service relies
tributed algor
consistency, c
to-scaling, loa
coordination t
such algorithm
combining the
tem is a challe
must usually b
properly in a
addition, we h
to invent algo
work hard to a
plexity, but the
the task remain
Complexity
ity of human
and operation
the system cou
tion of data, o
contracts on w
pend. So, befo

Images sources: Wikipedia