Model Checking

One-Slide Summary

- **Model Checking** is an algorithmic method in software engineering used to verify if a finite-state model of a system meets specific requirements.
- **Formal Verification** refers to the process of using mathematical techniques to prove or disprove the correctness of a system concerning a formal specification.
- **Formal Methods** utilize mathematical techniques to specify, develop, analyze, and verify software and hardware systems.
- **Theoretical Aspects of Software Engineering (TASE)** involve studying and applying mathematical and logical foundations to understand, model, and enhance software engineering processes and systems.

Model Checking ⊂ Formal Verification ⊂ Formal Methods ⊂ TASE

Learning Objectives: by the end of today’s lecture, you should be able to…

1. (*Knowledge*) Review the foundations of software engineering
2. (*Value*) Understand the concept of formal methods and its relation to software engineering
3. (*Skill*) Review formal verification and model checking

Overview

- Motivation
- Background and Basic Concepts
- Formal Verification and Model Checking
- Abstract (Semantic) Models
- Linear Time Logic (LTL)

Motivation

What is Model Checking?

- Model checking is a formal verification technique in software engineering that algorithmically verifies if a finite-state model of a system satisfies a given specification, usually expressed in temporal logic.
- It systematically explores all possible states of the system to ensure correctness and identify potential errors.

The potential of model checking

- Model checking is particularly valuable for safety-critical systems because it can rigorously verify that these systems meet their specifications and do not exhibit undesirable behaviors.
- Model checking helps ensure the reliability and safety of these systems by exhaustively exploring all possible states and transitions to detect errors early in the development process.

Model checking Prospects

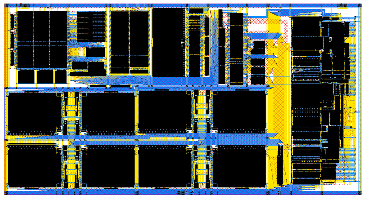
- Theoretically speaking, model checking of large language models (LLMs) may even be possible!!!
- The operational semantics of an LLM may be represented as a transition system.
- In this framework, each state represents a specific configuration of the model's parameters and memory, while transitions correspond to the model's responses to inputs or internal updates.

Too big a transition system! Why?

- Parameter Space: LLMs have billions of parameters, each of which can take on a wide range of values.
- Input Combinations: The variety of possible inputs (words, sentences, contexts) further multiplies the number of potential states.
- Internal Memory: The model's internal memory and context tracking add another layer of complexity.
- Given these factors, the total number of states can be on the order of (10^{100}) or more, depending on the specific architecture and application of the LLM.

Symbolic Model Checking

- Symbolic Model Checking using data structures such as OBDDs has been able to verify transition systems with more than 10^{120} states.



10^{120} states

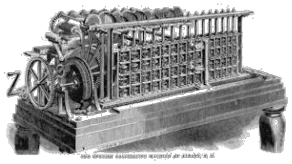
10^{51} atoms

https://sciencenotes.org/how-many-atoms-are-in-the-world/
https://education.jlab.org/qa/mathatom_05.html

  - Model Checking =

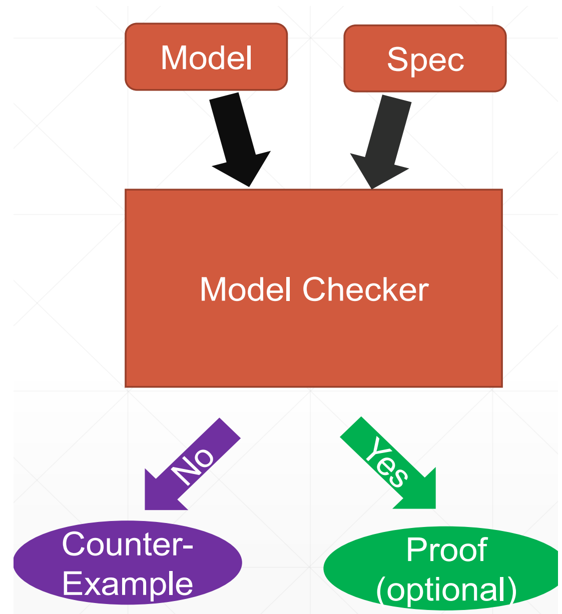
    —mechanical, push-button technology

    —performed without human intervention



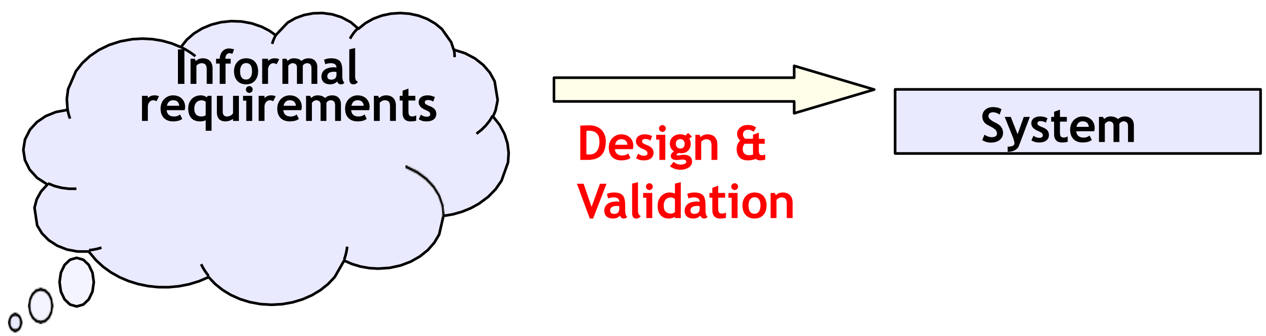
Model Checking Approach

- An approach for verifying the temporal behavior of a system.
- Primarily fully automated (“push-button”) techniques.
- Model
  - Representation of the system
  - Need to decide the right level of granularity.
- Specification
  - High-level desired property of a system
  - Considers infinite sequences.
- PSPACE-complete w.r.t. size of the specification model and linear w.r.t. size of the transition system model.



Conventional software engineering

- From requirements to a software system
  - apply design and validation methodologies
  - code directly in a programming language
  - validation mainly via testing, code walkthroughs, etc.



But my program works!

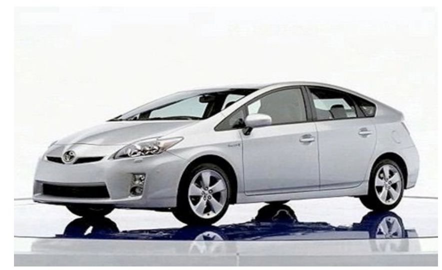
- True, there are many successful large-scale complex computer systems…
  - online banking, electronic commerce
  - information services, online libraries, business processes
  - supply chain management
  - mobile phone networks

- Yet many new potential application domains with far greater complexity and higher expectations
  - automotive drive-by-wire
  - medical sensors: heart rate & blood pressure monitors
  - intelligent buildings and spaces, environmental sensors

- Learning from mistakes is costly…

Toyota Prius

- Toyota Prius
  - first mass-produced hybrid vehicle
- February 2010
  - software "glitch" found in anti-lock braking system
  - in response to numerous complaints/accidents

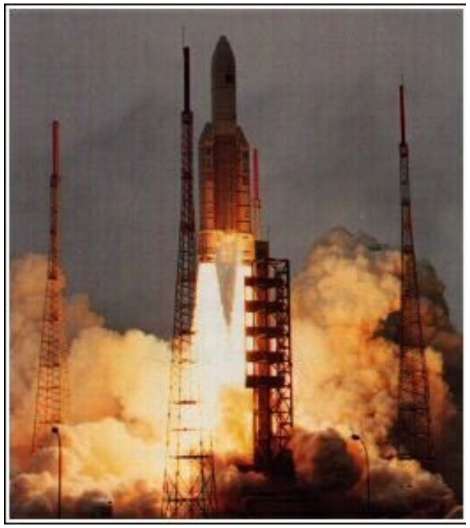


- Eventually fixed via software update
  - in total 185,000 cars were recalled, at a huge cost
  - handling of the incident prompted much criticism, bad publicity

Ariane 5

- ESA (European Space Agency) Ariane 5 launcher
  - shown here in maiden flight on 4th June 1996
- 37secs later self-destructs
  - uncaught exception: numerical overflow in a conversion routine results in incorrect altitude sent by the on-board computer

Expensive, embarrassing…



The London Ambulance Service

- London Ambulance Service computer-aided despatch system
  - Area 600sq miles
  - Population 6.8million
  - 5000 patients per day
  - 2000-2500 calls per day
- Introduced October 1992
- Severe system failure:
  - position of vehicles incorrectly recorded
  - multiple vehicles sent to the same location
  - 20-30 people estimated to have died as a result

Smart Vehicles

- Safety-critical systems
  - Airplanes
  - Space shuttles
  - Railways
- Expensive mistakes
  - Chip design
  - Critical software
- Want to guarantee safe behavior over unbounded time
- https://web.eecs.umich.edu/~movaghar/Software Model Checking-CACM2010.pdf

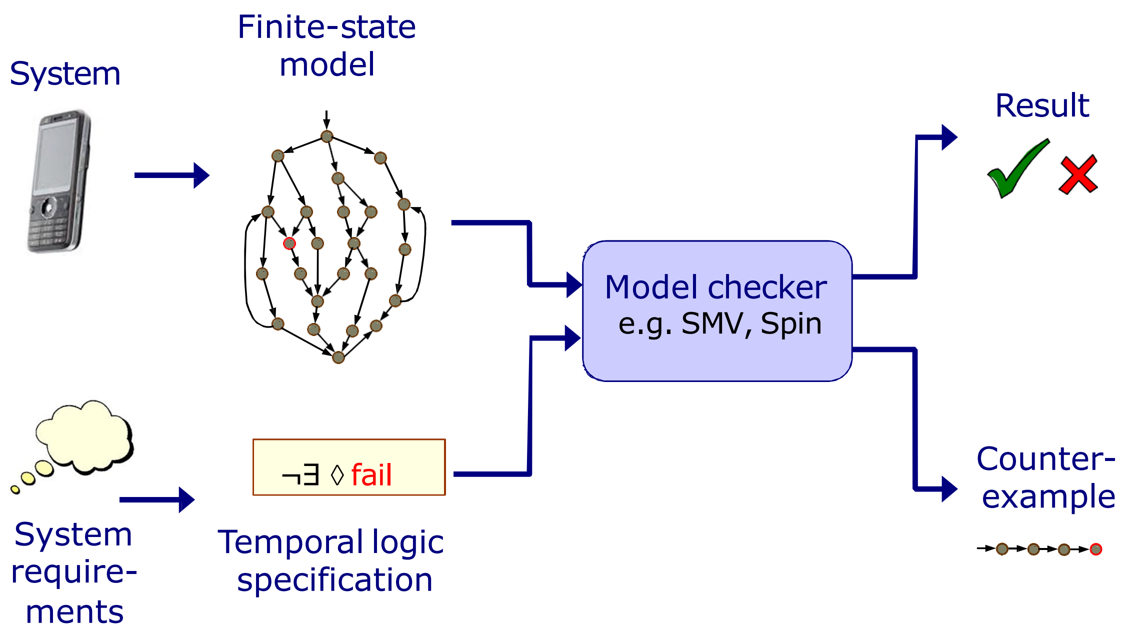

Smart vehicles

What do these stories have in common?

- Programmable computing devices
  - conventional computers and networks
  - software embedded in devices
    - airbag controllers, mobile phones, etc.
- Programming error direct cause of failure
- Software critical
  - for safety
  - for business
  - for performance
- High costs incurred: not just financial
- Failures avoidable…

Model Checking

Automated formal verification for finite-state models



3

Model checking in practice

- Model checking now routinely applied to real-life systems
  - not just “verification”…
  - model checkers used as a debugging tool
    - at IBM, bugs detected in arbiter that could not be found with simulations
- Now widely accepted in industrial practice
  - Microsoft, Intel, Cadence, Bell Labs, IBM,...
- Many software tools, both commercial and academic
  - NuSmv, PRISM, SPIN, SLAM, FDR2, FormalCheck, RuleBase, ...
  - software, hardware, protocols, …
- Extremely active research area
  - 2008 Turing Award won by Edmund Clarke, Allen Emerson, and Joseph Sifakis for their work on model-checking. https://web.eecs.umich.edu/~movaghar/Model Checking-Turing Award2007-CACM.pdf

Overview

- Motivation
- Background and Basic Concepts
- Formal Verification and Model Checking
- Abstract (Semantic) Models
- Linear Time Logic (LTL)

Background and Basic Concepts

Software Engineering

- IEEE, in its standard 610.12-1990, defines software engineering as the application of a systematic, disciplined, which is a computable approach for the development, operation, and maintenance of software, that is, the application of engineering to software.

What is Software?

- We will present a philosophical definition of software (hardware) as follows.

- Accordingly, we find that:
  - Software is more complex than hardware.
  - Software is a complex system.

Philosophical Definition of Hardware

- Hardware refers to the tangible, physical components of a computer system.
- Philosophically, hardware can be seen as the material substrate that provides the necessary infrastructure for computational processes.
- It is the physical embodiment of the machine's capabilities, constrained by its material properties and design.

Philosophical Definition of Software

- Software, on the other hand, is intangible.
- It consists of the instructions and data that tell the hardware how to perform tasks.
- From a philosophical perspective, software represents a computer system's logical and functional essence.
- It is the abstract set of rules and instructions that govern the behavior of the hardware, enabling it to perform complex operations.

Philosophical Distinction

- Tangibility: Hardware is defined by its physical presence, while software is independent of a particular physical form.
- Functionality: Hardware's primary purpose is physical functions, whereas software's is to execute logical functions, manipulating symbols and data.
- Malleability: Hardware is relatively difficult to change once manufactured, while software can be easily modified and updated.

Brain versus Mind

- The philosophical distinction between the hardware and the software is often compared to the relationship between the brain and the mind in the philosophy of mind, where:
  - the brain is the physical hardware and
  - the mind is the software or the set of processes and functions that the brain performs.

The theoretical aspects of software engineering (TASE)

- The theoretical aspects of software engineering focus on the underlying principles and formal methods that guide the development, verification, and maintenance of software systems.

Key Areas of TASE (1/2)

- Formal Methods: Techniques like model checking, theorem proving, and formal verification to ensure software correctness and reliability.

- Program Semantics: Understanding the meaning of programs through formal languages and logic.

- Type Systems: Ensuring program correctness by defining and enforcing rules about how data types are used.

Key areas of TASE (2/2)

- Abstract Interpretation: A theory used to analyze programs by approximating their behaviors.

- Automata Theory: The study of abstract machines and the problems they can solve, which is fundamental in designing compilers and interpreters.

Formal Methods in Software Engineering

- Formal methods in software engineering are mathematically rigorous techniques used for the specification, development, analysis, and verification of software and hardware systems.
- These methods involve the use of formal languages, logic, and mathematical models to ensure the correctness and reliability of software systems.

Formal Methods: Key Areas

- Formal Specification: Creating precise and unambiguous descriptions of software behavior and properties.

- Formal Verification: Using mathematical proofs to verify that software meets its specifications.

- Model Checking: Automatically verifying finite-state models of software against desired properties.

Formal Methods: Applications

- Formal methods are particularly valuable in safety-critical and security-critical systems, such as avionics and medical devices, where reliability is paramount.
- Formal methods can be considered as research works in software engineering. This research contributes to advancing the theoretical foundations of software engineering and enhancing practical applications.

https://github.com/ligurio/practical-fm

https://web.eecs.umich.edu/~movaghar/Formal Methods Manifesto 2023.pdf

Amazon

Amazon is actively involved in research related to formal verification, particularly through its Amazon Web Services (AWS) division. Here are a few notable examples:

- Cryptographic Software: Amazon's Automated Reasoning group has used formal verification to improve the efficiency and security of cryptographic algorithms, such as RSA, on their Graviton2 chips. This ensures that the cryptographic software behaves correctly and securely.
- Amazon s2n: This is an open-source implementation of the TLS (Transport Layer Security) protocol used by many Amazon services. Formal verification is continuously applied to s2n to ensure its correctness and security throughout its lifecycle.
- Cloud Infrastructure: Formal verification tools are used within AWS to enhance the security of its cloud infrastructure, helping to secure both the infrastructure itself and the customers using it.

Microsoft

Microsoft is actively involved in research related to formal verification across various projects and divisions. Here are a few notable examples:

- The Research in Software Engineering (RiSE) group at Microsoft focuses on formal methods, automated reasoning, and proof-oriented programming to ensure the correctness and security of their systems.
- VeriSol: This is a formal verification tool developed by Microsoft Research for verifying smart contracts written in the Solidity programming language. VeriSol helps ensure the correctness and security of smart contracts used in Azure Blockchain.
- Verus: This project focuses on creating a practical foundation for systems verification. It aims to eliminate bugs at compile time, ensuring that software is correct before it ships.
- Practical System Verification: This initiative explores methods to make formal verification more practical and scalable for system software. It addresses the challenges of verifying complex systems with minimal developer effort.

Meta (formerly Facebook)

Meta is involved in research related to formal verification.

- One notable project involves applying formal verification to microkernel inter-process communication (IPC). This research uses Iris, a concurrent separation logic implemented in the Coq proof assistant, to verify queue data structures used for IPC in an operating system under development at Meta. The project has successfully identified and corrected bugs, leading to more reliable and efficient code.
- Meta's efforts in formal verification are part of a broader initiative to enhance the reliability and security of its software systems. This includes using formal methods to verify the correctness of algorithms and improve the overall robustness of their infrastructure.

Airbus

Airbus is actively involved in research related to formal verification, particularly for its avionics software. Here are some key points:

- Avionics Software: Airbus has been integrating formal verification techniques into the development process of avionics software since 2001. These techniques include abstract interpretation, theorem proving, and model-checking.
- DO-178B Compliance: The formal verification methods used by Airbus comply with the stringent requirements of the DO-178B standard, which governs the development of avionics software.
- Collaborations: Airbus collaborates with academic and industrial labs, such as ONERA (the French aerospace lab), to advance formal verification methods and their application in critical embedded systems.
- Tools and Techniques: Airbus has developed and transferred several formal verification tools to its operational teams, including Caveat, aiT, and Stack analyzer, which are used to achieve DO-178B verification objectives.

NASA

NASA is deeply involved in research related to formal verification, particularly to ensure the safety and reliability of its aerospace systems. Here are some key areas of their work:

- Flight Critical Software: NASA has applied formal verification techniques to flight critical software, such as the Flight Control Systems (FCS) used in aircraft. This involves using formal methods to verify the behavior of system components and ensure they meet safety requirements.
- Langley Formal Methods Program: The Formal Methods group at NASA's Langley Research Center develops and maintains the NASA PVS Library, which includes a wide range of formal verification tools and frameworks. These tools are used for verifying air traffic systems, fault-tolerant protocols, and other critical systems.
- Autonomous Systems: NASA also explores formal verification approaches for autonomous robotic systems. This includes specifying and verifying the behavior of autonomous systems to ensure they operate safely and reliably in space missions..

Tesla

Tesla is involved in research related to formal verification, particularly in the context of its autonomous driving systems.

- Formal verification methods are used to ensure the safety and reliability of the software that controls Tesla's vehicles. This includes verifying that the software behaves correctly under all possible conditions, which is crucial for the development of safe and reliable autonomous vehicles.
- Tesla's Autopilot and Full Self-Driving (FSD) systems rely heavily on advanced software engineering and formal methods to validate the complex algorithms that enable autonomous driving. This research helps in identifying and mitigating potential risks, ensuring that the systems operate safely in real-world scenarios.

xAI

- xAI is actively involved in research related to formal verification.
- According to recent reports, xAI plans to incorporate formal verification techniques into its AI models.
- This approach aims to ensure that the code generated by their models, such as the Grok language model, is free from bugs and adheres to specified safety and performance criteria.
- Formal verification is a mathematical method used to prove the correctness of systems, and its application in AI can significantly enhance the reliability and trustworthiness of AI-generated outputs. This is particularly important for safety-critical applications where errors can have serious consequences.

SpaceX

- SpaceX is involved in research related to formal verification, particularly to ensure the safety and reliability of its spacecraft and rocket systems. Formal verification methods are crucial for validating the complex software that controls these systems, ensuring they perform correctly under all possible conditions.
- SpaceX's software engineering teams use formal methods to verify the correctness of flight control systems, mission planning software, and other critical components. This rigorous approach helps prevent errors that could lead to mission failures, making it an essential part of their development process.

OpenAI

- Neural Theorem Proving: OpenAI has developed a neural theorem prover for the Lean proof assistant, which is used to solve formal mathematics problems. This involves using language models to generate proofs for formal statements, enhancing the reliability and correctness of mathematical proofs.
- Autoformalization: OpenAI has explored autoformalization, which is the process of translating natural language mathematics into formal specifications and proofs. This research aims to improve the accuracy and efficiency of formal verification by leveraging large language models.
- Improving Verifiability: OpenAI has published reports on mechanisms to improve the verifiability of AI systems. These tools help developers provide evidence that AI systems are safe, secure, fair, and privacy-preserving.

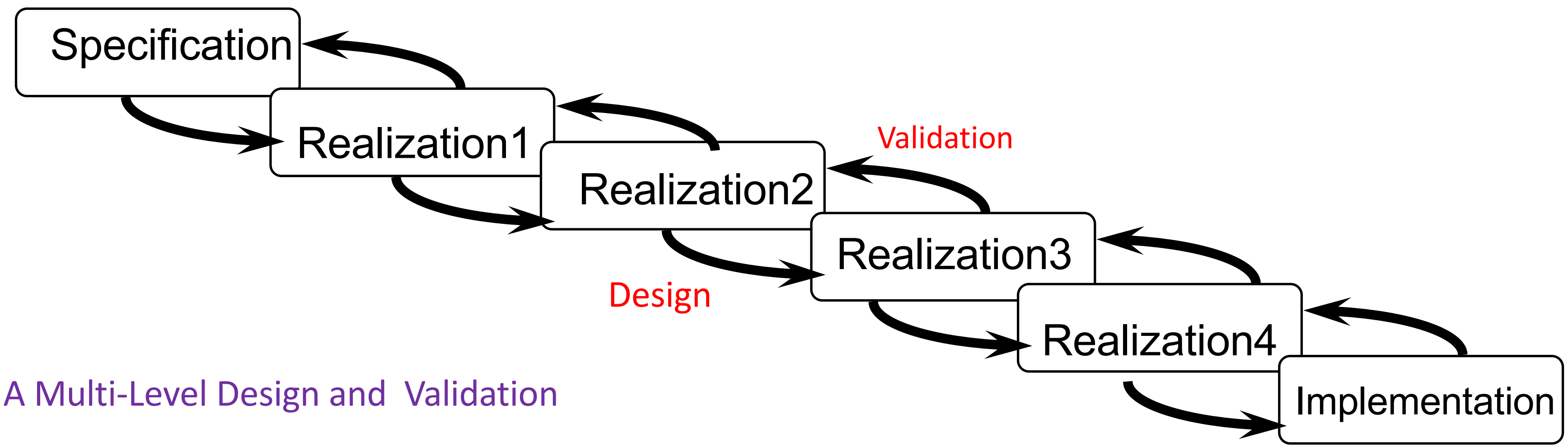
Google

Google is actively involved in research related to formal verification. Here are some key areas of their work:

- Towards Making Formal Methods Normal: Google Research has published work on integrating formal methods into developers' existing practices and workflows. This research aims to increase the adoption of formal verification by making it more accessible and practical for everyday software development.
- Formal Verification Techniques: Google uses formal methods to verify critical software systems. This involves modeling system requirements using specification languages and validating these models with tool support to ensure consistency and early verification.
- Automated Reasoning: Google's Automated Reasoning team focuses on developing tools and techniques for formal verification to improve the reliability and security of software systems. This includes work on verifying cryptographic protocols and other critical components.

Design and Validation

- A design is a process of getting a (more detailed) realization from a given specification.
- Validation is a process of ensuring that a realization satisfies its specification.



A Multi-Level Design and Validation

- An implementation may be viewed as the lowest level of realization.

Design

- Design is a process of getting a realization from a given specification.
- The design of a complex system may happen on many levels.
- An implementation may be viewed as the lowest level of realization.

Validation

- Validation is a process of ensuring that a realization satisfies its specification.

Validation Methods

- Validation has three main methods:
  - Formal Verification
  - Evaluation
  - Testing

Formal Verification

- Formal Verification is a mathematical method to prove that a realization satisfies its specification.

Evaluation

- Evaluation is a method for finding how well a system behaves.

Testing

- Testing is a method of proving that a realization does not satisfy its specification.

Integrated Validation

- Testing, Formal Verification, and Evaluation are usually complementary.

Evaluation Methods

- Measurement
- Analytical Modeling
- Simulation Modeling
- Hybrid Modeling

So, why not only test?

- Testing only shows the presence of bugs, not their absence!

Verification and Formal Verification

- In software engineering, verification and formal verification are crucial processes, but they differ significantly in their approaches and objectives.
- Both methods are essential for ensuring a system is reliable, functional, and meets user needs.
- They are often used together to provide comprehensive assurance of system quality.

Verification

- Verification is the process of ensuring that the software meets its specified requirements.
- It involves checking that software is built correctly according to the design and specifications.
- Verification aims to catch errors early in the development process and ensure that the software behaves as expected under specified conditions.

Formal Verification

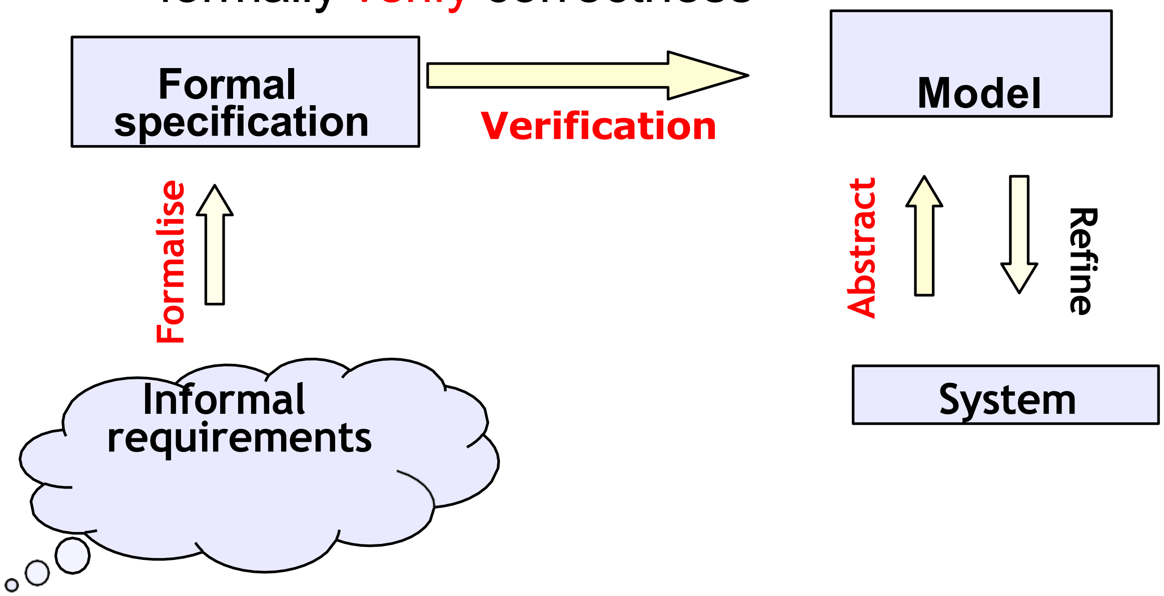
- Formal Verification, on the other hand, uses mathematical methods to prove the correctness of a system.
- It involves creating formal software models and using logical reasoning to verify that the software adheres to its specifications under all possible conditions.
- Formal verification provides a higher level of assurance because it can prove the absence of certain errors, rather than just finding them through testing.

Verification versus Formal Verifications

- Approach: Verification relies on testing and reviews, while formal verification uses mathematical proofs and models.
- Scope: Verification checks the software against specific scenarios, whereas formal verification aims to prove correctness under all possible scenarios.
- Assurance: Formal verification offers a higher level of confidence in the correctness of the software, as it can mathematically guarantee certain properties.

Formal verification

- From requirements to formal specification
  - formalize specifications, derive a model
  - formally verify correctness



Overview

- Motivation
- Background and Basic Concepts
- Formal Verification and Model Checking
- Abstract (Semantic) Models
- Linear Time Logic (LTL)

Formal Verification and Model Checking

What are Formal Methods?

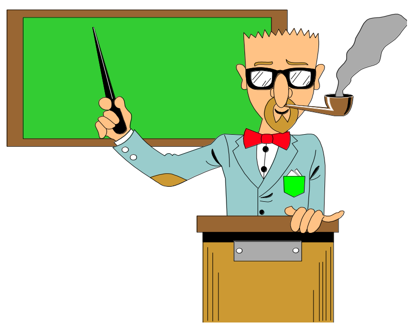
- Techniques for analyzing systems, based on some mathematics.
- This does not mean that the user must be a mathematician.
- Some of the work is done informally, due to complexity.

Formal Methods

- Mathematically-based techniques for describing properties of systems
- Provide framework for
  - Specifying systems (and thus the notion of correctness)
  - Developing systems
  - Verifying correctness
    - Of implementation w.r.t. the specification
    - Equivalence of different implementations
- Reasoning is based on logic
  - Amenable to machine analysis and manipulation

Why aren’t FMs used more?

“Formal methods can revolutionize development!”



“Formal methods are difficult, expensive, not widely useful and for safety-critical systems only”

Formal Verification

- Formal verification seeks to establish a mathematical proof that a system works correctly.

https://web.eecs.umich.edu/~movaghar/Principles of Model Checking-Book-2008.pdf

Formal Verification Steps

- A formal verification is done in three steps:
  - A system model to describe the system,
  - A specification model to describe the correctness requirement,
  - An analysis technique to verify that the system meets its specifications.

Some System Model

- Transition Systems (Automata)
- Communicating Sequential Processes (CSP)
- Reo
- (High-level) Programming Languages

- https://web.eecs.umich.edu/~movaghar/cspbook.pdf
- https://web.eecs.umich.edu/~movaghar/Reo Arbab 2003.pdf

Some Specification Models

- Propositional Logic
- First-order Logic
- Linear Temporal Logic (LTL)
- Computational Tree Logic (CTL)

Formal Verification Methods

- There are two major methods for formal verification:
  - Deductive Method
  - Model Checking

Deductive Method

- In the deductive method, the problem is formulated as proving a theorem in a mathematical proof system.

Model Checking

- In the method of model checking, the behavior of the system is checked algorithmically through an exhaustive search of all reachable states.

https://web.eecs.umich.edu/~movaghar/Model Checking-Turing Award2007-CACM.pdf

https://web.eecs.umich.edu/~movaghar/Model Checking-Q&A-Turing Award2007.CACM.pdf

https://en.wikipedia.org/wiki/Model_checking

View of Model Checking in Theoretical Settings

- Theoretical Aspects of Software Engineering
  - Formal Methods
    - Formal Verification
      - Model Checking

Popular Model-Checking Tools

- NuSMV
- PRISM

https://nusmv.fbk.eu/index.html

Companies Using NuSMV

- Airbus: Utilizes NuSMV for verifying the correctness and safety of their avionics system.
- Intel: Employs NuSMV in the verification of hardware designs to ensure reliability and performance.
- Siemens: Uses NuSMV for verifying industrial automation systems and ensuring they meet safety standards.
- NASA: Applies NuSMV in the verification of critical software systems used in space missions.

Companies Using PRISM

- Google: Utilizes PRISM for verifying the reliability and performance of their systems, particularly in areas involving probabilistic models.
- Microsoft: Employs PRISM in their research and development to ensure the correctness and reliability of software systems.
- IBM: Uses PRISM for formal verification of complex systems, ensuring they meet required performance and reliability standards.

Overview

- Motivation
- Background and Basic Concepts
- Formal Verification and Model Checking
- Abstract (Semantic) Models
- Linear Time Logic (LTL)

Abstract (Semantic) Models

Abstract (Semantic) Models

- A prerequisite for model checking is to provide a model of the system.
  - We introduce transition systems as abstract (semantic) models to represent hardware and software systems.
  - Using the Structural Operation Semantics (SOS) method, we can define the operational semantics of any model, including all (high-level) programming languages.

What are Transition Systems?

- Transition Systems (TSs) are abstract (semantic) models that are the operational semantics of many models, including all (high-level) programming languages.
- TSs are directed graphs where nodes and edges represent states and transitions, respectively.

States of a Transition System

- A state describes information about a system at a certain moment of its behavior:
  - The current color of a traffic light.
  - The current values of all program variables + the program counter.
  - The current value of the registers together with the values of the input bits.

Transitions of a Transition System

- Transitions specify how the system evolves from one state to another.
  - A switch from one color to another (for traffic light).
  - The execution of a program statement.
  - The change of the registers and output bits for a new input.

Structural Operation Semantics

- The transition relation of TS(Model) is defined using the so-called SOS notation:

$$\frac{\text{premise}}{\text{conclusion}}$$

  - This implies if the proposition above the “solid line” holds, then the proposition under the fraction bar holds as well.
  - If the premise is a tautology, the rule is called an axiom.

Operation Semantics

- The operational semantics of many models, including all (high-level) programming languages, are defined using SOS rules.

- https://web.eecs.umich.edu/~movaghar/SOS-1.pdf
- https://web.eecs.umich.edu/~movaghar/SOS-2.pdf

Overview

- Motivation
- Background and Basic Concepts
- Formal Verification and Model Checking
- Abstract (Semantic) Models
- Linear Time Logic (LTL)

Linear Time Logic (LTL)

What is Linear Temporal Logic (LTL)?

- Linear Temporal Logic (LTL) is a type of modal temporal logic used to describe sequences of events or states over time.
- LTL is widely used to formally specify safety-critical properties of hardware and software systems.
- For example, it can be used to ensure that a system will never reach an undesirable state or that a certain condition will eventually be met.

LTL Syntax

- LTL formulas are built using a set of propositional variables, Propositional operators (like ¬, ∨, ∧), and temporal operators.

Temporal Operators:

- X (Next): A condition will be true at the next state.
- F (Eventually): A condition will be true at some point in the future.
- G (Globally): A condition will always be true.
- U (Until): One condition will be true until another condition becomes true.

Example

- LTL allows for the specification of the relative order of events. However, it does not support any means to refer to the precise timing of events.
  - “The car stops once the driver pushes the brake”.
  - “The message is received after it has been sent”.

LTL Semantics

- LTL formulas are evaluated over infinite sequences of states (often called paths).

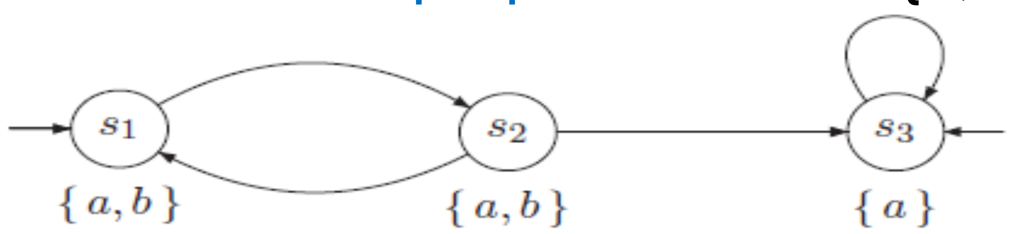
- A path satisfies an LTL formula if the formula holds for the entire sequence of states.

infinitely often and eventually forever

- By combining G and F, new temporal modalities are obtained:
  - GFa describes the property stating that at any moment j there is a moment l ≥ j at which a-state is visited. Thus a-state is visited infinitely often.
  - FGa expresses that from any moment j, finally only a-state is visited. Thus a-state is visited eventually forever.

LTL Example (1/2)

- **Example**: Consider the Transition System (TS) below with the set of atomic propositions AP={a,b}



  - TS $\models$ Ga
  - TS $\not\models$ X(a$\wedge$b)
  - TS $\models$ G($\neg$b $\rightarrow$G(a$\wedge\neg$b))
  - TS $\not\models$ b U (a$\wedge\neg$b)

LTL Example (2/2)

- **Example**: Properties for mutual exclusion problem:
  - Safety property states that two processes P_1 and P_2 never simultaneously have access to their critical section: $G(\neg crit_1 \vee \neg crit_2)$.
  - Liveness property stating each process P_i is infinitely often in its critical section: $(GFcrit_1) \wedge (GFcrit_2)$.