# Software Testing and Validation

A.A. 2025/2026

Corso di Laurea in Informatica

## Basic Notions

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

# General Info for This Class

- Software Testing and Verification is an elective course for the Informatica Bachelor Degree
- Lecturer: Igor Melatti
- Where to find these slides and more:
  - `https://igormelatti.github.io/sw_test_val/20252026/index.html` (Italian)
  - `https://igormelatti.github.io/sw_test_val/20252026/index_eng.html` (English)
  - also on MS Teams: "DT0758: Software Testing and Validation (2025/26)", code **86obv1d**
- 2 classes every week, 2 hours per class

# Rules for Exams

- The exam consists in working on a project and discuss it
  - teams of at most 3 students are allowed for projects
- Project: perform testing and validation of a given software
  - each team may choose one among the ones selected by lecturer
  - each team will have to discuss its project with slides
  - pre-evaluation is possible and encouraged
    - our last lessons will be dedicated to projects ongoing work discussion

# Verification Problem

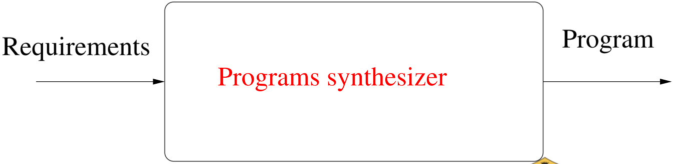
- Dates back to computer science origins
  - of course, not only in computer science
- Generally speaking: we have built something, is it what we actually wanted to build? does it accomplishes its tasks?
  - I do not want to get tired standing up, I build a chair
  - am I actually not tired any more? or at least, less than before?
  - does the chair crash if I sit for too much time?
  - does it crash if I increase my weight?
  - could I have built the chair better (more comfortable, with less materials, ...)?
- The same holds for software
  - but also for hardware, or combinations hardware+software

# Utopia!

1. Suppose you want to write a software fulfilling some given requirements
   - given an array $A$, sort $A$ in a non-decreasing way
   - given a graph $G = (V, E)$ and two nodes $u, v \in V$, decide if there exists a path from $u$ to $v$
   - build the data base for a library
   - manage an airport
   - etc.
2. Let us try to write the corresponding requirements
   - $\forall 1 \leq i \leq n-1 \; A[i] \leq A[i+1]$
   - $\exists u_1, \ldots, u_n$ s.t. $u_1 = u \wedge u_n = v \wedge \forall 1 \leq i \leq n-1 \; (u_i, u_{i+1}) \in E$?
   - It is possible also for the remaining cases, though it is more complicated

# Utopia!

- Suppose you have an automatic program synthesizer (generator)
  - a special program which takes *requirements* as input
    - must be described in some *formal* way, i.e., using an unambiguous mathematical language
  - ... and outputs a correct-by-construction program which fulfills the input requirements



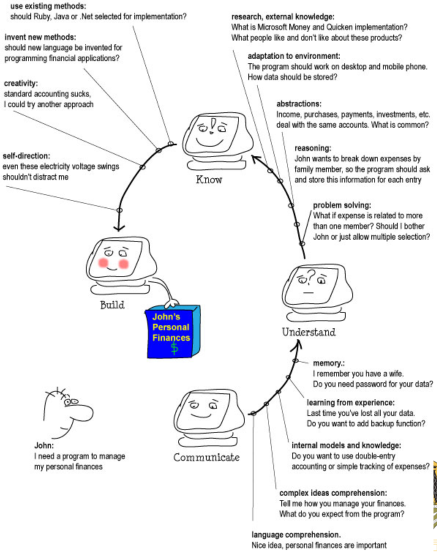

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

# Utopia!

- All efforts are in making the program generator correct, efficient and effective
- It outputs correct-by-construction programs
  - if I say "give me a program sorting arrays", then I obtain a program which *never* fails
  - i.e., given any array (input instance), it outputs always the correct sorted array (corresponding output)
- Verification problem does not exist
  - or it shifts on the requirements: did we write the requirements we actually wanted?
  - validation: we will be back on this

# Instead, the Reality



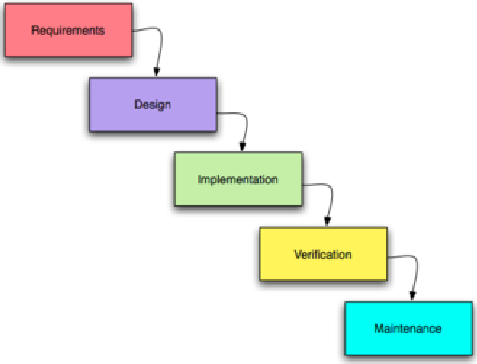
# Instead, the Reality

- Do you need to build a software? then, you will have to do it *ad hoc*
    - *totally* general approaches to build program generators cannot exist
    - it is easy to see that building a program generator is an undecidable problem
- Of course, you can rely on libraries, methodologies, etc, but...
- ... there is no guarantee that the starting requirements are met by the final software
    - e.g., if you implement an iterative program to sort arrays, but you forget to increment the index, the starting requirements will not be met
    - more subtle errors may be very difficult to find

- So you need a *verification* phase
  - for simple cases like sorting, it is sufficient to perform it in the end
  - for more complex cases, verification must be performed also during developing phase
  - for very complex/important cases, verification must be performed also *before* developing the software
- Verification goal is to find *errors*, if any
  - for our purposes, an error is a violation of the requirements
  - some requirements are present since the beginning, some other may add up later

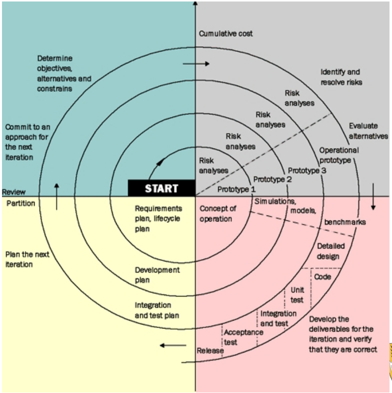
# Software Verification

- Software Engineers are well aware of the problem
- All software design processes include one or more verification phases
  - though it may be simply called *test* o *testing*



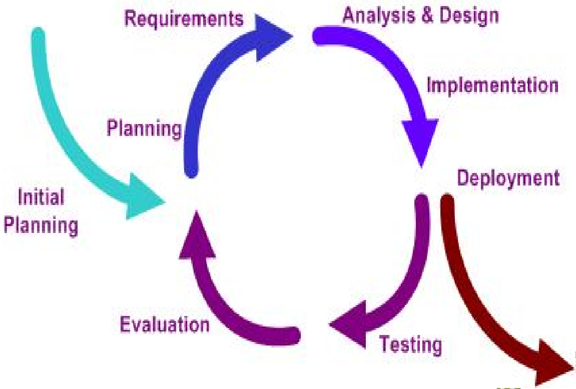
# Software Verification

- Software Engineers are well aware of the problem
- All software design processes include one or more verification phases
  - though it may be simply called *test* o *testing*



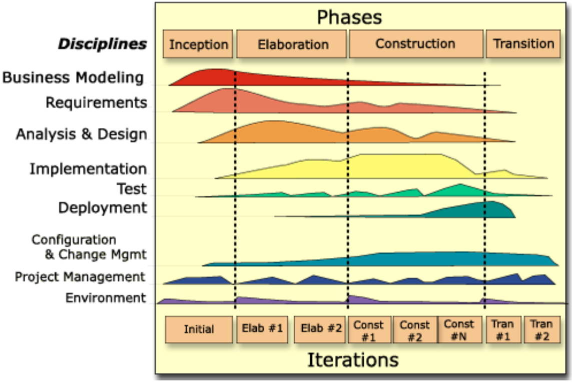
# Software Verification

- Software Engineers are well aware of the problem
- All software design processes include one or more verification phases
    - though it may be simply called *test* o *testing*



# Software Verification

- Software Engineers are well aware of the problem
- All software design processes include one or more verification phases
  - though it may be simply called *test* o *testing*



# *Systems* Verification

- We have been speaking of software, but all we said holds for *any computer-based system*
- Hardware
  - digital circuits
  - microprocessors
- Embedded Systems
  - tiny *dedicated* computer inside bigger systems
  - typically, either *controllers* or *monitors*
  - cars (ABS, ESC/ESP...), generic means of transportation, domestic electrical appliances (fridges, TVs, ...)
  - errors could be in hardware, software, both, or in the "communication" (interface) between hardware and software

Summing up:

1. start from requirements
2. develop some (partial or final) solution
   - you may "complicate" such steps at wish
3. *verify* that the current solution fulfills the starting requirements
   - if at least one error is discovered, correct it, going to step 2
   - you may need to correct the requirements, going to step 1
   - verification may (and should) be done during the intermediate developing steps
   - if no error, deploy solution

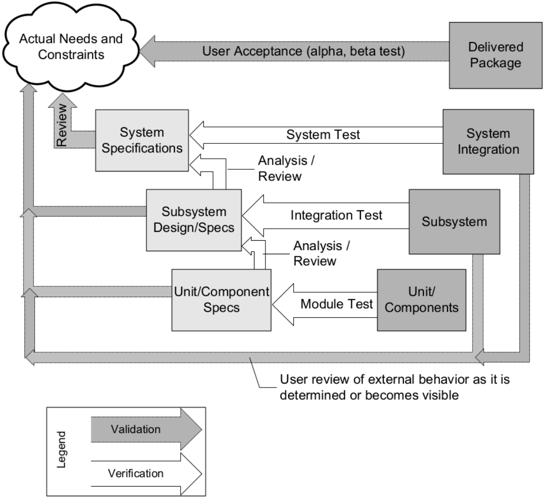
# Validation

- Verification and validation are often used as synonyms
- However, there is an important distinction between the two terms
  - validation involves final users "expectations"
  - verification is performed only keeping in mind the software requirements already collected
  - verification does not care whether requirements are what users want or not
- Validation is "did we built the right system?" $\rightarrow$ *useful* system
- Verification is "did we built the system right?" $\rightarrow$ *dependable* system

# Validation and Verification

- Requirements analysis vs. requirements specifications
  - requirements analysis: what (we understood that) the users want
  - requirements specification: the solution we propose for the requirements analysis
- Validation is about checking requirements anaysis
  - more focused on the overall requirements and the final code
- Verification is about checking requirements specifications
  - often with intermediate steps
- In this course, we will mainly focus on verification
  - though also validation will be treated

# Validation and Verification



# How Verification is Performed

Method number 1: *Testing*

1. you have the actual system (or a part of it)
2. you feed it with predetermined *inputs*
3. you check if *outputs* are the expected ones
   - "expected" w.r.t. the requirements
4. if there is one output different from the expected one, then we have an error
5. you correct it and start over again
   - restarting from the "highest" point where you made the correction
   - requirements, design, code

# How Verification is Performed

Method number 1 bis: *Simulation*

- two typical cases:
  - prototyping: you do not have the full code, but some simplified prototype may be built
    - feed inputs to the prototype instead of the actual software
    - especially useful to test designs (early testing)
  - you have the full code, but it is used to control/monitor of some physical system (*cyber-physical systems*)
    - the simulator is for such physical system: it accepts the same inputs and provides the same outputs of the physical system
    - connect the software to such simulator as it was the real system
    - proceed as in "normal" testing by feeding inputs and observing outputs
    - you might also use a prototype for the (control/monitor) software and a simulator for the physical system for early testing

# How Verification is Performed

Cyber-physical systems: why this methodology?

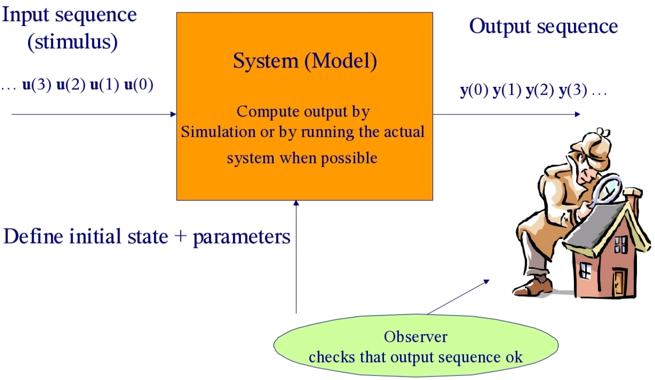
- Must check if they work *before* connecting to the physical part
  - or, even worse, build it
  - at least, the most common/easy errors must be ruled out
- If you have a controller for a plane, you do not directly test it on an actual plane, a simulator of the plane is used
  - only when tests on the simulator are ok you move to test on the actual plane
  - if the simulator says the plane is crashed, it is less severe than an actual plane crashing
- It is not a matter of safety only: it might also be an economical problem
  - e.g., testing on microprocessors must use some simulator before, as "writing" on silicon is expensive
  - e.g., if you are building a new airplane also basing on its controller, you must know if there are problem in the design

# How Verification is Performed: Errors Correction

- This might not be easy: testing typically only *triggers* errors
- Then, you might have to reproduce the error in some smaller scale
- Then, you have to understand where the problem is and what causes it
  - requirements? architecture? design? single point in the code? an intricated flow in the code?
- Then, design and implement the actual correction
- In this course, we only deal with error triggering

# How Verification is Performed

## An approximate answer
## BUG HUNTING: Testing + Simulation



# How Verification is Performed

- Both testing and simulation may be performed in refined ways
- In fact, the *testing plan* (the predetermined sequence of inputs) may be computed using dedicated algorithms so that *coverage* is maximized
  - we will get back soon on this concept
- This is the most challenging and important step for such techniques

# Testing and Simulation: Pro and Cons

Pro

- (Relatively) easy to implement
  - easier than the other methods we will consider here
- Largely used in industry
  - in most cases, testing and/or simulation are the *only* verification methods used

Cons

- They can prove that a system *has* errors, but cannot prove that a system *does not have* errors
- Cannot be used to prove generic formal properties
- The coverage of the "input space" is low
- Errors are frequently detected when it is too late

# Testing and Simulation: Cons

They can prove that a system *has* errors, but cannot prove that a system *does not have* errors

- If an error is detected, then the system must be corrected, happy to have discovered it
- Otherwise, *we cannot conclude anything*
- That is, we cannot say that the system is error-free
- In fact, having not be able to spot errors does not imply that there are no errors

# Testing and Simulation: Cons

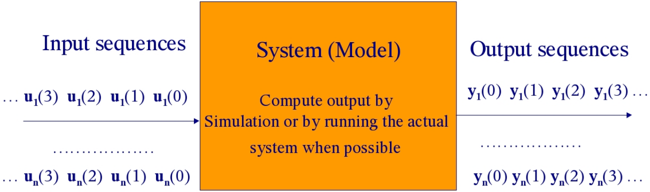
Cannot be used to prove generic formal properties

- This is a consequence of the previous slide
- As an example: in an operating system, is it true that mutual exclusion is enforced for 2 given processes?
- In order to test such a property you would have to modify the system itself
  - so that the output contains something like "propriety violated" or "'property ok"
- But even in this case, we cannot draw a formal statement on the validity of the property
- Again, not finding a violation does not imply there are no violations

# Testing and Simulation: Cons

The coverage of the "input space" is low

- A successful testing phase should consider "all what may happen" to the system in a real-world environment
- This would need too much tests or simulations




- The $n$ in the figure may easily be $10^6$ and more; outputs must also be checked

# Testing and Simulation: Cons

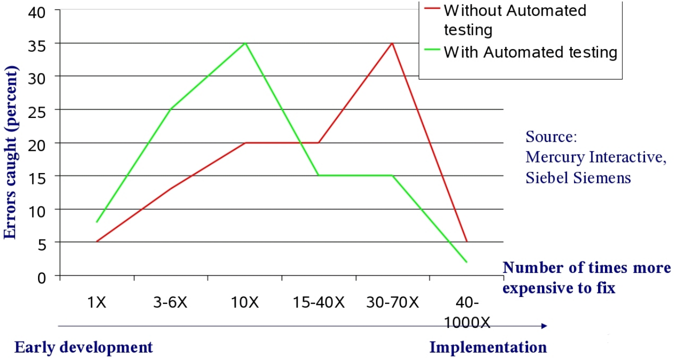
The coverage of the "input space" is low

- This also has another bad consequence
- Testing and simulation find the "easy" errors
  - the most frequent ones
  - i.e., those that are caused by many (different) input sequences
- Instead, *corner cases* usually go undetected
  - i.e., errors that are caused by a few (or even single) input sequences are usually not found

# Testing and Simulation: Cons

Errors are frequently detected when it is too late

- This is a consequence of the previous point: you need many tests to get a reasonable coverage and discover possible corner cases
- The later an error is found, the more expensive the correction



# *Formal Verification*

- To solve the above underlined problems, we should consider *all* inputs
- That is, all possible system *evolutions*
  - of course, testing and simulation only consider *some* evolutions: those "activated" by inputs chosen by the testing plan in use
- A possible way to do this is to prove a dedicated theorem, stating that the system is correct for all inputs
- For sorting, this could be done (and it is actually done in Algorithms textbooks...)
- For other cases (e.g., microprocessor design), it would be too difficult or time consuming
- Thus, techniques of *formal verification* have been developed

# *Formal Verification* Methods

- A set of (heterogeneous) techniques which make possible the impossible
- That is, algorithms able to generate and analyze *all* system evolutions
  - so, they provide a *mathematical certification* of correctness (not achievable with testing/simulation)
  - also for generic properties, like mutual exclusion
- Actually, the problem of verifying a given system w.r.t. a given property is *undecidable*
  - the property to be verified may be: is this system always terminating?
- So, there will be some (acceptable in many cases) limitations

# Is Formal Verification Useful?

- There are many techniques available for formal verification
- Applying any of these techniques is usually much more difficult than testing/simulation
  - both in terms of personnel and notions required
- So, why to do this?
- Because there are many cases in which testing/simulation simply *are not enough*
  - for both economic and safety reasons

# Is Formal Verification Useful?

- Safety-critical systems: failures may affect humans
  - public transport software controllers (if an automatic pilot of an airplane has a failure...)
  - trains crossing
  - ABS for cars
  - ...
- For most of such systems, formal verification is mandatory by law
  - ESA (European Space Agency)
  - IEC (International Electrotechnical Commission)

# Is Formal Verification Useful?

- Mission-critical systems: failures cause huge economic losses
  - automatic space probes
  - logistics
  - communication networks
  - microprocessors
  - ...
- Internal company regulations often make formal verification mandatory as well

# Is Formal Verification Useful?

- Also for systems which are neither safety nor mission critical: there are economic motivations to use formal verification
- Using testing/simulations, errors are eventually discovered
- The problem is that they may be found *late*
  - this is a consequence of the low coverage issue
- So late, that often errors are found *after* the system has been deployed, i.e., when it is already used by its final users
  - for, e.g., a *word processor*, it is annoying, but we are somewhat used to software updates to fix bugs
  - this is not always possible or easy
    - e.g., a legacy software out of support

# Is Formal Verification Useful?

- Hardware circuits: to "write" a circuit on silicon is the most expensive part of the developing process
- So, finding an error after having written the circuit entails a huge economic loss
- This also holds for other systems, when the developing process is lengthy
- In fact, finding a late error may cause going again through preceding developing phases
  - less competitivity on the market
  - for both being late and for augemented costs

# Is Formal Verification Useful?

- Some famous errors in safety-critical systems
  - 20/7/1969: on the Apollo 11, the driving computer fails multiple times during the final descent on the Moon
    - all ok because the large support team on Earth finds out that the error may be ignored
  - 26/9/1983, URSS believes USA have launched 5 nuclear weapons
    - no 3rd WW only because a Russian official finds it strange there are only 5 missiles
    - all due to a software bug in recognizing false negatives
  - 1985-1987: Therac-25, computer system to treat cancer through rediations
    - many patients due to too high radiations
    - the error was afterwards tracked to a “race condition” among concurrent processes

# Is Formal Verification Useful?

- Some famous errors in mission-critical systems
  - 1962: Mariner 1 automatic space probe (80 M$)
    - the dash sign for negative numbers is missing ("the most expensive dash in history")
    - resulting trajectory is completely wrong
    - the support team blows the probe to avoid it hitting something on ground
  - 1990: AT&T network failure
    - just one code line wrong in one telephone exchange
    - for hours, 60000 users are unable to make calls
  - 1990: another space probe, Ariane 5 (500 M€)
    - overflow in converting numbers from 64 to 16 bits (!)
    - due to reuse of Ariane 4 software

# Is Formal Verification Useful?

- Some famous errors in mission-critical systems (continued)
  - 1994: Intel Pentium computes wrong answers on some floating point errors (450 M$)
  - 2006: Airbus A380 internal wires
    - errors in the software controlling wiring
    - all design process have to be restarted from scratch
    - extremely huge economic losses
  - 2010: Toyota Prius ABS
    - error "glitch" in the ABS controller
    - 185,000 cars recalled for updating
    - also bad publicity

# Is Formal Verification Useful?

- A should-be-famous error in mission-critical systems: Needham-Schroeder protocol
  - public-key authentication protocol, designed in 1978
  - widespread use in many systems for decades
  - initiated a large body of work on the design and analysis of cryptographic protocols
- After 17 years of usage, an error was (manually) discovered in 1995 by Lowe
- In 1996, Lowe showed that, using formal verification, it would have been easy to immediately detect the error
  - more in detail, by using model checking
- Other examples are in `https://spinroot.com/spin/success.html`

# Is Formal Verification Useful?

The most recent case: CrowdStrike vulnerability

# Is Formal Verification Useful?

The most recent case: CrowdStrike vulnerability

> **La "schermata blu della morte" che ha bloccato i computer in tutto il mondo: ecco cosa è successo. "Più pesante del Millennium Bug"**
>
> di Diego Longhin

# Is Formal Verification Useful?

The most recent case: CrowdStrike vulnerability

The New York Times

**Global Tech Outage** | What We Know | When Tech Fails | More Flights Canceled | Passengers Still Struggling | Guard Against Scams

## Chaos and Confusion: Tech Outage Causes Disruptions Worldwide

Airlines, hospitals and people's computers were affected after CrowdStrike, a cybersecurity company, sent out a flawed software update.

Share full article | 951

# Summing Up

- Testing and simulation are the most used verification tools
  - most companies (especially for software) use *only* these tools
  - easier and cheaper to use
  - at least one between testing and simulation are *always* performed
- For mission critical or safety critical systems, formal verification methods must be used
  - more difficult to be applied
  - may provide a mathematical certification for the system correctness
  - only applied when budget allows it

# Formal Verification Methodologies: a Classification

There are two macro-categories:

- *Interactive methods*
- *Automatic methods*

# Formal Verification Methodologies: a Classification

There are two macro-categories:

- *Interactive methods*
  - as the name suggests, not (fully) automatic
  - human intervention is typically required
  - in this course, we do not deal with such techniques
- *Automatic methods*

# Formal Verification Methodologies: a Classification

There are two macro-categories:

- *Interactive methods*
  - as the name suggests, not (fully) automatic
  - human intervention is typically required
  - in this course, we do not deal with such techniques
- *Automatic methods*
  - only human intervention is to *model* the system

# Formal Verification Methodologies: a Classification

There are two macro-categories:

- *Interactive methods*
  - as the name suggests, not (fully) automatic
  - human intervention is typically required
  - in this course, we do not deal with such techniques
- *Automatic methods*
  - only human intervention is to *model* the system
- There also exist hybridations among the two categories

# *Interactive* Methods

- Also called *proof checkers*, *proof assistants* or *high-order theorem provers*
- Tools which helps in building a mathematical proof of correctness for the given system and property
- Pros
  - virtually no limitation to the type of system and property to be verified
- Cons
  - highly skilled personnel is needed
  - both in mathematical logic and in deductive reasoning
  - needed to "help" tools in building the proof

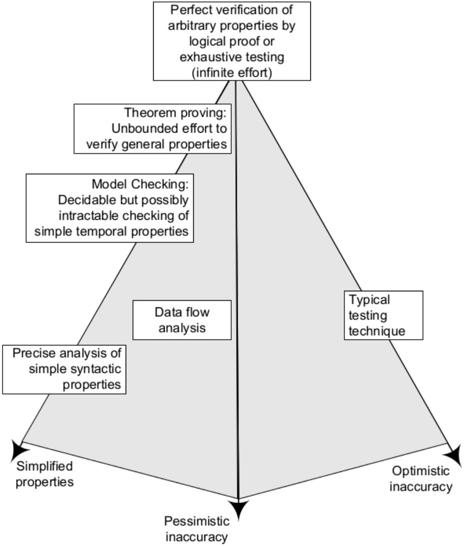
# *Interactive* Methods

- Used for projects with high budgets
- For which the automatic methods limitations are not acceptable
  - used, e.g., to prove correctness of microprocessor circuits or OS microkernels
- Some tools in this category (see `https://en.wikipedia.org/wiki/Proof_assistant`):
  - HOL
  - PVS
  - Coq

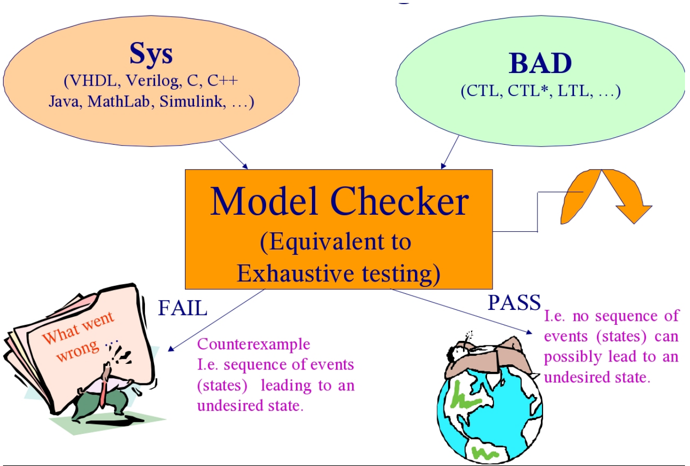
# *Automatic* Methods

- Commonly dubbed *Model Checking*
- Model Checking software tools are called *model checkers*
- There are some tens model checkers developed; the most important ones are listed in `https://en.wikipedia.org/wiki/List_of_model_checking_tools`
- Many are freely downloadable and modifiable for research and study purposes
- Research area with many achievements in over 30 years

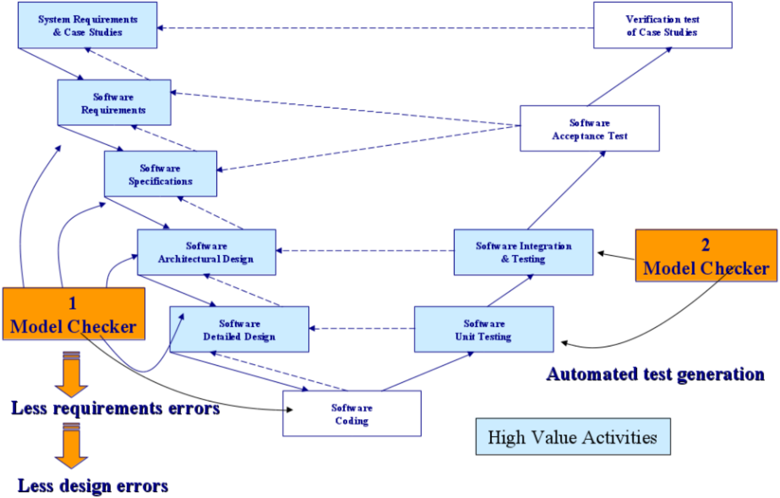
# Verification Tradeoffs



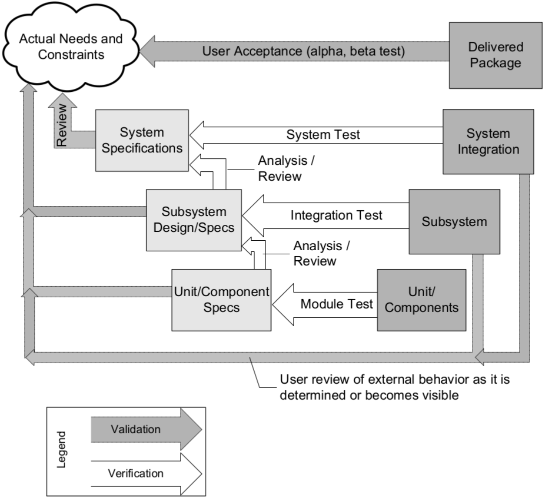
# The Model Checking Dream



# The Model Checking Dream



# Also Keep This in Mind



# Actual Model Checking

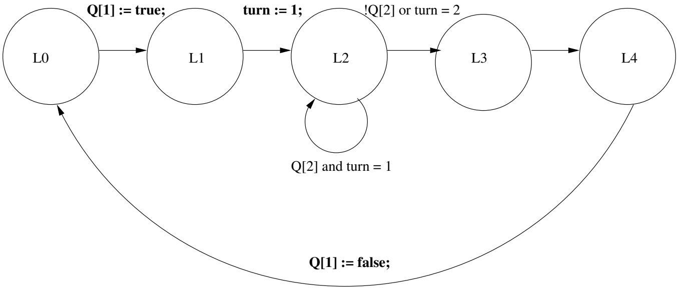
- In order to have this computationally feasible, we need a strong assumption on the system under verification (SUV)
- I.e., it must have a *finite number of states*
  - *Finite State System* (FSS)
- In this way, model checkers "simply" have to implement reachability-related algorithms on graphs
- Such finite state assumption, though strong, is applicable to many interesting systems
  - that is: many systems are actually FSSs
  - or they may be approximated as such
  - or a part of them may be approximated as such

# What Is a *State*?

- There are many notions of "state" in computer science
- Model checking states are *not* the ones in UML-like state diagrams
- Model checking states are similar to operational semantics states
- That is: suppose that a system is "described" by $n$ variables
- Then, a state is an assignment to all $n$ variables
  - given $D_1, \ldots, D_n$ as our $n$ variables domains, a state is $s \in \times_{i=1}^{n} D_i$

# What Is a *State*: Example

- We have two identical processes accessing a shared resource
  - in the figure below, $i, j$ denote the two processes
  - the well-known Peterson algorithm is used

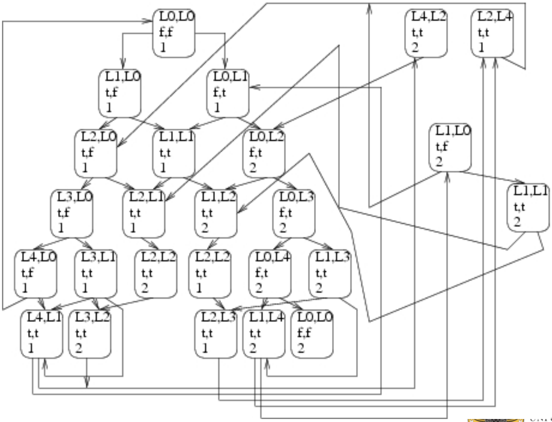


# What Is a *State*: Example

- The 5 "states" in the preceding figure are actually *modalities*
- From a model checking point of view, they correspond to *multiple* (i.e., sets of) states
- To see which are the actual states, let us model this system with the following variables:
  - $m_i$, with $i = 1, 2$: the modality for process $i$
  - $Q_i$, with $i = 1, 2$: $Q_i$ is a boolean which holds iff process $i$ wants to access the shared resource
  - `turn`: shared variable

# What Is a *State*: Example

- Thus, the resulting model checking states are the following:



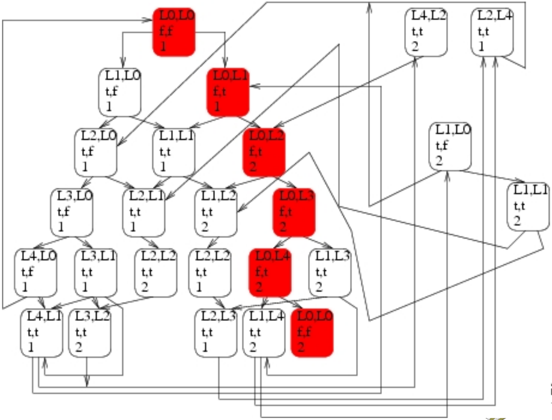
# What Is a *State*: Example

- There are 25 *reachable states*
  - assuming state $\langle L0, L0, f, f, 1\rangle$ as the starting one
- All *possible* states are 200
  - there are 3 variables with two possible values (the 2 variables `Q`, plus the `turn` variable) and 2 variables (`P`) with 5 possible values, thus $2^3 \times 5^2$ overall assignments
- The `L0` modality for the first process encloses 6 (reachable) states

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

# What Is a *State*: Example



# What Is a *State*: Example

- There are 25 *reachable states*
  - assuming state $\langle L0, L0, f, f, 1 \rangle$ as the starting one
- All *possible* states are 200
  - there are 3 variables with two possible values (the 2 variables `Q`, plus the `turn` variable) and 2 variables (`P`) with 5 possible values, thus $2^3 \times 5^2$ overall assignments
- The `L0` modality for the first process encloses 6 (reachable) states
- No need of guards on transitions!

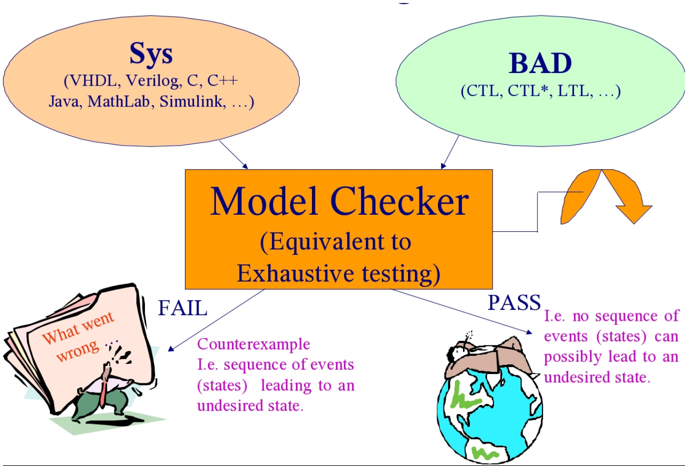
# From State Diagrams to Model Checking

- The UML-like state diagram is often useful to write the model
  - as we will see, this will depend on the model checker *input language*
- It is the model checker task to extract the global (reachable) graph as seen before
- And then analyze it

## Is Model Checking Important?

- ESA, NASA e IEC require most of their project to be model checked
- Important companies have dedicated laboratories for Model Checking
  - hardware: Intel, IBM, SUN, NVIDIA
  - software: IBM, SUN, Microsoft
- Many universities have research groups
  - USA: MIT, CMU, Austin, Stanford...
  - very close collaboration with companies
- The 3 “inventors” of Model Checking received Touring Award in 2007:
  - E. A. Emerson, E. M. Clarke, J. Sifakis

# Model Checking Usage



# Model Checking Usage

3 steps:

0. Choose the model checker $M$ which is most suitable to the SUV $\mathcal{S}$ (and the property $\varphi$)
1. Describe $\mathcal{S}$ in the input language of $M$
2. Describe the property $\varphi$
3. Invoke the model checker and wait for the answer
   - OK $\Rightarrow \mathcal{S} \models \varphi$
   - FAIL $\Rightarrow$ counterexample
     - correct the error (it may happen that $\mathcal{S}$ or $\varphi$ must be corrected instead...) and go back to step 3
   - OutOfMem or OutOfTime
     - adjust system parameters (or the description of $\mathcal{S}$)

# Model Checking Usage

- Most used for *reactive systems*
  - always executing systems:
    - monitors: warns if something bad happens
    - controllers: avoids that something bad happens
    - services: wait for requests and serve it
  - more in general, concurrent execution of processes/threads with shared memory/messages exchange
  - errors may occur because of interactions/interleaving between different processes/threads
- Not good for standalone (1-process) programs
  - e.g., sorting an array or perform BFS of a graph
  - for such systems, testing can be complemented with theorem proving (or with manual proof derivation)
  - of course, budget must be taken into account

# Model Checking: Pro and Cons

Pro

- Same guarantees of proof checking
- But requiring less "mathematics" and "computer science" knowledge

Cons

- Computational Complexity
  - causing "OutOfMem" and "OutOfTime": *State Explosion Problem*
- You check a model of the system, not the actual system
  - though in some cases models can be automatically extracted from the system
- Useful only for multi-process/thread software

# State Explosion Problem: Why?

- With some semplification, all Model Checking algorithms are essentially like this:
  1. Extract, from the description of the SUV $\mathcal{S}$, the *transition relation* of $\mathcal{S}$
  2. Compute the *reachable states* (*reachability*)
  3. Check if $\varphi$ holds in all reachable states
- All steps may be computationally heavy, but let us focus on the reachability
  - see mutual exclusion example
- If $\mathcal{S}$ is described by $n$ (binary) variables, then the number of reachable states is $O(2^n)$

# State Explosion Problem: Why?

- Such complexity cannot be avoided in the most general case
- Theoretically speaking, (LTL) Model Checking is P-SPACE complete
  - CTL Model Checking is in P, but as we will see this does not make things better
- There are several model checking algorithms, depending on the “type” of $\mathcal{S}$
  - each checker has its “preferred” SUVs

# Model Checking Algorithms

There are 3 categories:

- Explicit
- Implicit (symbolic)
- SAT-based

# Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
- SAT-based

# Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
  - dedicated data structures are used to represent sets of states
  - very good for digital hardware
- SAT-based

# Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
  - dedicated data structures are used to represent sets of states
  - very good for digital hardware
- SAT-based
  - many problems may be theoretically rewritten as SAT, but in model checking this works pretty well also in practice
  - software model checking

# Model Checking Algorithms

There are 3 categories:

- Explicit
  - each reachable state is separately stored
  - very good for communication protocols
- Implicit (symbolic)
  - dedicated data structures are used to represent sets of states
  - very good for digital hardware
- SAT-based
  - many problems may be theoretically rewritten as SAT, but in model checking this works pretty well also in practice
  - software model checking
- Proof checker ibridations
  - not completely automatic, but better than proof checkers

# Model Checking-Related Problems

- Controllers generators
  - particular case for the program synthesizer seen in the beginning
  - controllers are software modules which sends digital commands to some physical device
  - in some cases, they may be built automatically, using algorithms similar to those of Model Checking
- Probabilistic Model Checking
  - verification of stochastic processes
- Stochastic Model Checking
  - verification outcome is correct with high probability

# From Model Checking to Testing

- Not all software is mission- or satefy-critical
  - actually, most software do not fall in such categories
- Moreover, testing is required also for such systems
  - not all features may be checked through formal verification
- Hence, testing is at least as important as model checking and similar techniques
  - early 2000s estimate: software failures cost US economy nearly 60 billion$ per year
  - early 2000s estimate: at least 22 billion$ per year could be saved by applying proper software testing
- No general frameworks exist, but we have some general "best practises"
  - we will cover them in this course

# What If We Use AI?

- Many powerful AI tools have been recently developed and made accessible:
  - general-purpose: ChatGPT, DeepSeek, Claude, Perplexity, ...
  - programming specific: Copilot, Llama, ...
- How they affect what we see in this course?
- We have to first consider how they can be used when developing/implementing/testing some software
  - directly generate software implementations from specifications
  - given a software, tell me if there are errors
  - ~~given a software, directly perform testing~~ AI LLMs refuse to run software
  - given a software, list some interesting test cases
  - given software specifications, output a description for some model checking tool

# What If We Use AI: Software Generation


- Temptation: if it is output by AI, it is correct-by-construction
  - the verification problem simply disappears
- This is extremely far from reality
  - thus impractical for mission or safety critical software
- Finding errors in AI-generated software requires (human) developers to first understand it

# What If We Use AI: Software Analysis

- Provide the source code to AI and ask if it can spot any errors
- If an error is found, mostly good
  - AI answers may always contain errors, but a (human) developer should be able to check if the error is a false negative or not
- If an error is not found, again *no guarantee* that there are no errors
- However, asking a check to AI may be a good idea as a start of the verification

# What If We Use AI: Test Cases Generation

- Provide the source code to AI and ask test cases as output
  - also specifications (for the whole software or for some parts) may be provided: white-box testing
- Again, AI answers may always contain errors
  - in this case, it may be that output test cases are not well-formed, i.e., they do not consider all inputs
  - if they are well-formed, their *coverage* (roughly, capability of finding errors, if any) could be worse than what a human tester could produce
  - especially if the methodologies explained in this course are adopted...

# What If We Use AI: Model Checking Specification

- Provide the software specifications to AI and ask a specification for some model checking tool as output
- Like the generating software point, no guarantee of "correctness"
    - i.e., of representing the system correctly
    - or, if it does, of being actually usable

# We Will See Theory...

- A Kriepke structure is a 4-tuple: $\langle S, I, R, L \rangle$
- Formulas satisfiability: $\pi \models \varphi \mathbf{U} \psi$ iff $\exists j \in \mathbb{N}\ \forall 0 \leq i < j \pi(i) \models \varphi \wedge \pi(j) \models \psi$
- $\mu$-calcolus, e.g.: $R(x) = \mu Z[I(x) \vee \exists x'[N(x', x) \wedge Z(x')]]$
- Algorithms on graphs, hash tables, OBDDs...

# ...and Practice

- We will examine the most important model checkers, also considering the source code
  - often very well written
  - in order to delay state explosion as much as possible
  - good way to learn how to code
- SUVs modeling examples
- Software testing best practices, with examples

# Roadmap

1. Modeling systems with the Murphi model checker
2. Kripke structures and algorithms inside Murphi: Model Checking of invariants
3. LTL and CTL properties
   - safety and liveness
4. CTL Model Checking algorithms
5. LTL Model Checking with SPIN
6. CTL Model Checking with NuSMV
7. Bounded Model Checking with NuSMV
8. Testing (starting from November)
   - granularity
   - techniques
   - best practises

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica