Software Validation and Verification

Lorenzo Ceragioli

IMT school Lucca
lorenzo.ceragioli@imtlucca.it

GianLuigi Ferrari

UniPI
gian-luigi.ferrari@unipi.it

About Me

- Assistant Professor at IMT Lucca (RTD-A)
  - Systems Security Modelling and Analysis SySMA research unit
  - Bachelor, Master degree and PhD in Pisa with Degano and Galletta
- Research Interests
  - Verification of concurrent and interactive quantum systems (with Gadducci at UniPI)
  - Formal methods for computer security (with Galletta at IMT Lucca)
- Contacts
  - Mail: lorenzo.ceragioli@imtlucca.it
  - Page: lceragioli.github.io
  - Office time: Send me an email and we will schedule it!

IMT SCUOLA ALTI STUDI LUCCA

IMT SCUOLA ALTI STUDI LUCCA

SySMA Research Unit

Mirco Tribastone

Full Professor
SySMA Head
Modeling and Simulation, Software Performance Engineering, Computational Methods

Alessandro Armando

Full Professor (IMT and University of Genova)

Cybersecurity, Computer security

Alessandro Betti

Assistant Professor

Machine Learning, Computer Vision, Lifelong Learning

Lorenzo Ceragioli

Assistant Professor

Formal Methods, Software Security, Quantum Communication and Computing

Gabriele Costa

Associate Professor

Cybersecurity, Penetration Testing, Formal Methods, Software Verification, Vulnerability Assessment

Letterio Galletta

Assistant Professor

Software Security, Software Verification, Formal Methods, Programming Languages

Emilio Incerto

Assistant Professor

Software Performance Modeling and Control, Layered Queueing Networks, Autoscaling, Cloud Computing

Cosimo Perini Brogi

Assistant Professor

Formal Methods, Mathematical Foundations of CS, Proof Theory, Certified Programming, Type Theory, Non-classical Logics

Fabio Pinelli

Associate Professor

Data Science, Machine Learning, Spatio-temporal Machine Learning

Simone Soderi

Assistant Professor

Physical Layer Security, 6G Security: Optical communications, Covert channels, Security in critical infrastructure systems

Course Outline

- Introduction to model checking ~2/3 of the course
  - Lectures + Exercise Sessions
  - The Subject for the written exam
  - We will follow **Principles of Model Checking** by Christel Baier and Joost-Pieter Katoen
- Seminars on State-of-the-Art research ~1/3 of the course
  - Guest Lecturers will present advanced topics
  - You can select the subject for your seminar
  - Research papers will be given as suggested reading

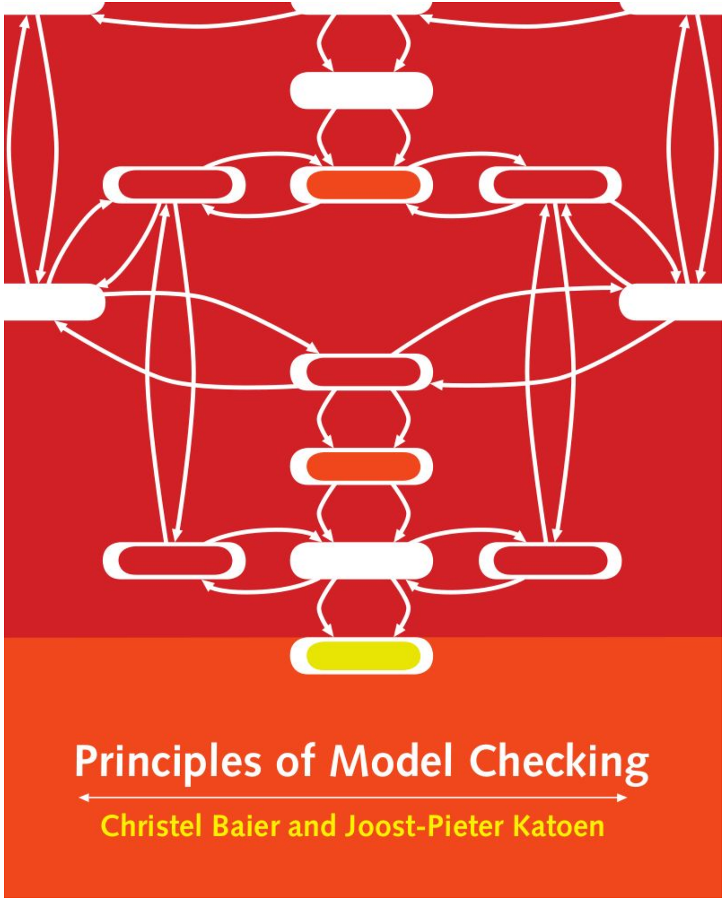
Exam

1. **A written exam on model checking** (we will see the syllabus shortly)
    - You must get at least 18/30L before scheduling the seminar

2. **A seminar** (followed by questions) presenting the content of a research paper on one of the advanced topics introduced during the last part of the course
    - A list of topics and related papers will be given
    - Recall to introduce the needed background (but you can assume the audience knows the basics of model checking and the course prerequisites)
    - Around 30 minutes plus questions

Course Material

- We will follow the book **Introduction to model checking** by Baier and Katoen, chapters 1 to 6 (see the errata corrige)
- We will also frequently use their slides
- Exercises sheets and solutions
- Everything but the book can be found at my page lceragioli.github.io/ (**announcements section** in case of room changes, cancelled lessons etc)
- Papers from the seminars



Course Prerequisites

- Automata and language theory
- Algorithms and data structures basics
- Computability and complexity theory
- Mathematical logic

Model Checking Course Syllabus

- **Modelling Systems**
  - Transition systems and program graphs
  - Modelling Concurrent Systems
- **Linear Time Properties**
  - Invariants, Safety, Liveness and Fairness
  - Checking regular safety properties
  - Checking omega regular properties with Büchi automata
- **Linear Time Logics**
  - Positive Normal Forms
  - Fairness
  - Model checking LTL formulas
- **Branching Time Logics**
  - Computational Tree Logics
  - Comparison of the expressivity of LTL, CTL and CTL*
  - Model checking CTL and CTL* formulas

… it will make more sense after an introduction to model checking

What is System Verification?

“System verification amounts to establishing whether the system under consideration possesses certain properties.”

More time and effort on verification and validation than on construction

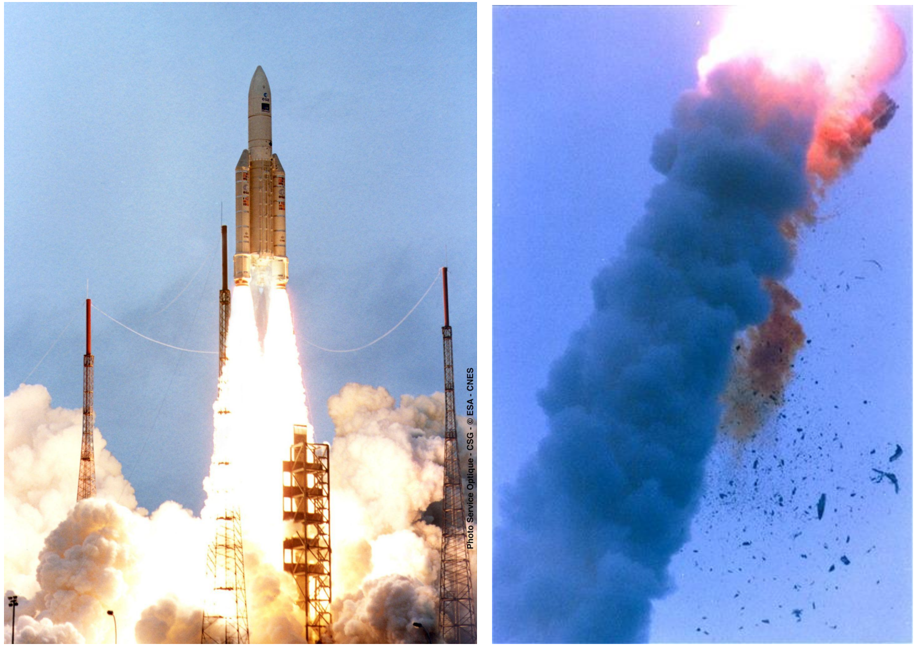
Verification = “are we building the thing **right**?”

Validation = “are we building the **right** thing?”

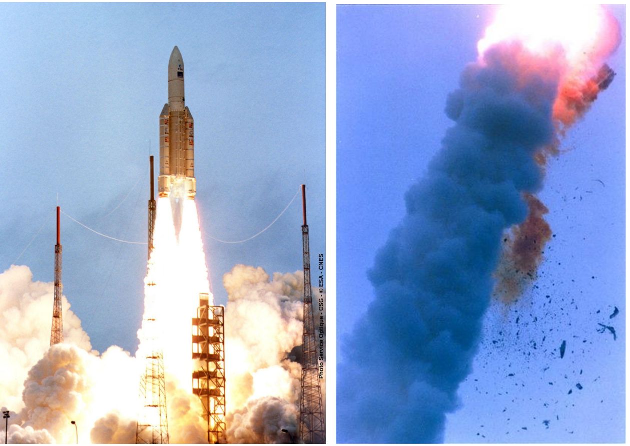
Note: Correctness is always relative to a specification

Because

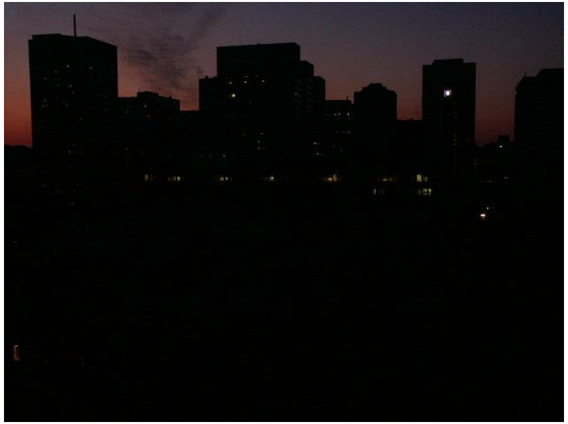
- The number of defects grows exponentially with the number of interacting system components (**concurrency**, nondeterminism)
- Some systems cannot be (easily) fixed after release
- Failures in critical systems may be catastrophic
- In catching software errors, the sooner is the better
- It is just about money and safety



Explosion of first Ariane 5 flight, 1996
(overflow while converting from 64-bit floating point to 16-bit signed integer)



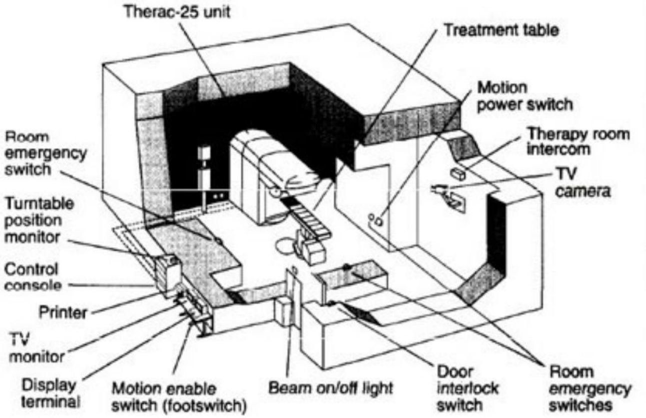
Explosion of first Ariane 5 flight, 1996
(Overflow during data conversion)



Northeast blackout, 2003
(Mishandled race condition)

DAO attack on Ethereum, 2016
(Reentrancy problem)

Pentium FDIV bug, 1994
(Missing values in a lookup table)



Therac-25 Radiation Overdosing, 1985-87
(Mishandled race condition)

Informal Approaches to System Verification

- **Peer review**
  - software inspection carried out by a team of engineers
  - static technique: manual code inspection
  - Subtle errors are hard to catch (e.g. concurrency)
- **Software simulation and testing**
  - take a model (simulation) or a realisation (testing)
  - stimulate it with certain inputs, i.e., the tests
  - observe reaction and check whether this is “desired”
  - number of possible behaviours is very large
  - unexplored behaviours may contain the fatal bug

Formal Methods: Applied mathematics for modelling and analysing ICT systems

Deductive Methods

Associate logical statements and derivation rules to program constructs, and derive a proof of the property for the system.

E.g. dependent types, proof assistants, Hoare logic …

Model-Based Methods

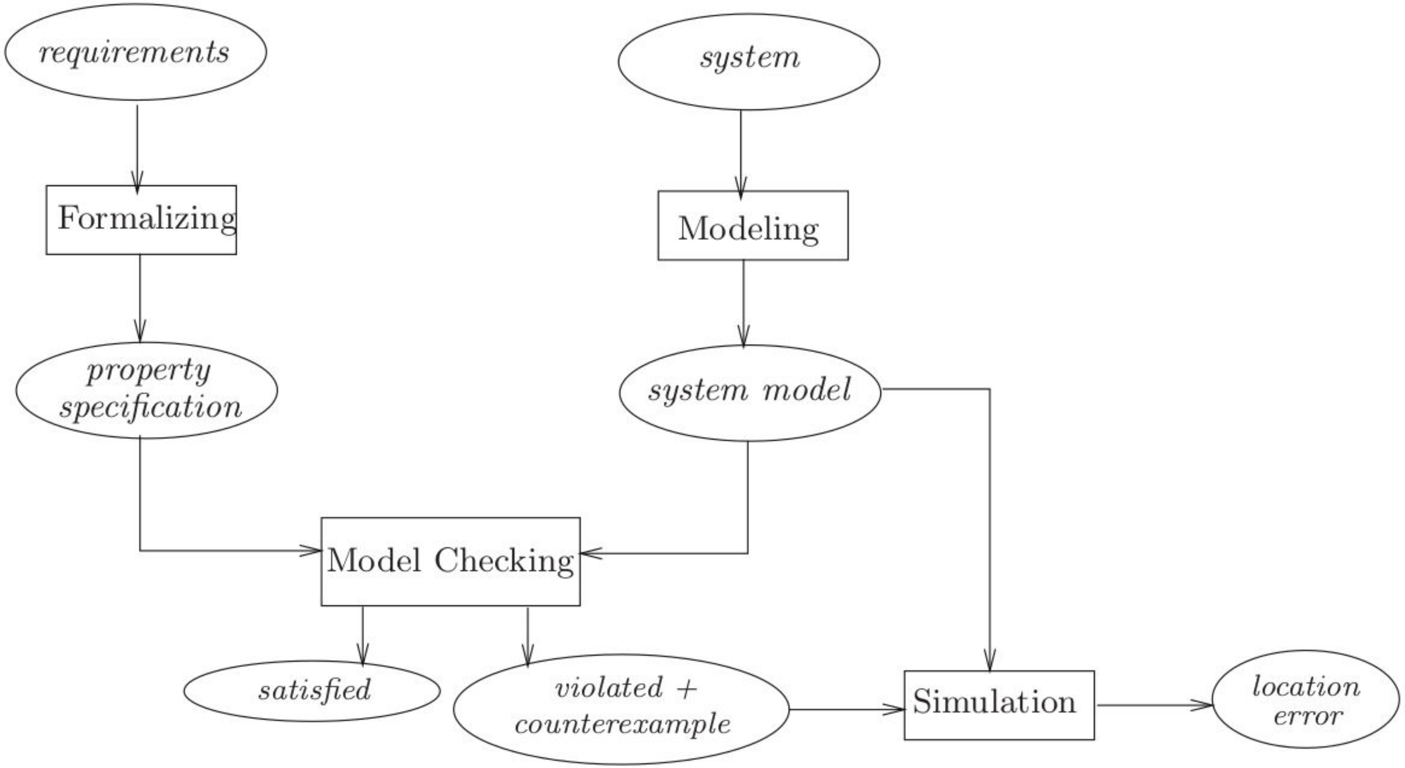
Generate and inspect a model describing the system behavior in a mathematically precise and unambiguous manner.

E.g. formal simulation and testing, **model checking** …

Model Checking

Model checking is an automated technique that,
given a finite representation of the behaviour of a system and a formal property,
systematically checks whether this property holds

Model Checking Approach Schema



Which Formal Model?

- Transition Systems
  - States in which the program may be
  - Propositions associated with states satisfying them
  - Transitions for representing state updates
  - Labels over transitions to represent interaction in a composable way

- Representing programs, possibly with multi-threads and communication

Modeling the System

- The model checker usually comes with a **model description language** (e.g. Promela for the SPIN model checker)
- The target system may be an **abstract entity**, like a cryptographic protocol
- Or it may be a **real system**, like a piece of code
- If the language of the target system has a **formal semantics** then correctness of the model can be formally proved
- Otherwise, correctness can only be "**corroborated** by experiments" though simulation

Formalizing the Requirements

- Usually by some **modal logic** (decidability/expressivity balance)
  - Modal operators such as "always", "eventually", "necessarily", "possibly"
  - $\Box P$, $\Diamond P$, with P a logical proposition
- **functional correctness** (does the system do what it is supposed to do?)
- **reachability** (is it possible to end up in a deadlock state?)
- **safety** ("something bad never happens")
- **liveness** ("something good will eventually happen")
- **fairness** (does, under certain conditions, an event occur repeatedly?)

(We should check consistency, otherwise model checking is useless)

Recall: Propositional Logic

$$\Phi \quad ::= \quad \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi$$

$$a \in AP$$

Model-based semantics

Interpretations $\mu : AP \rightarrow \{0,1\}$

$$\begin{array}{lll} \mu \models \text{true} & & \\ \mu \models a & \text{iff} & \mu(a) = 1 \\ \mu \models \neg\Phi & \text{iff} & \mu \not\models \Phi \\ \mu \models \Phi \wedge \Psi & \text{iff} & \mu \models \Phi \text{ and } \mu \models \Psi \end{array}$$

Deduction system

Based on proofs: inductively defined data structures (lists or trees) constructed according to the axioms and derivation rules.

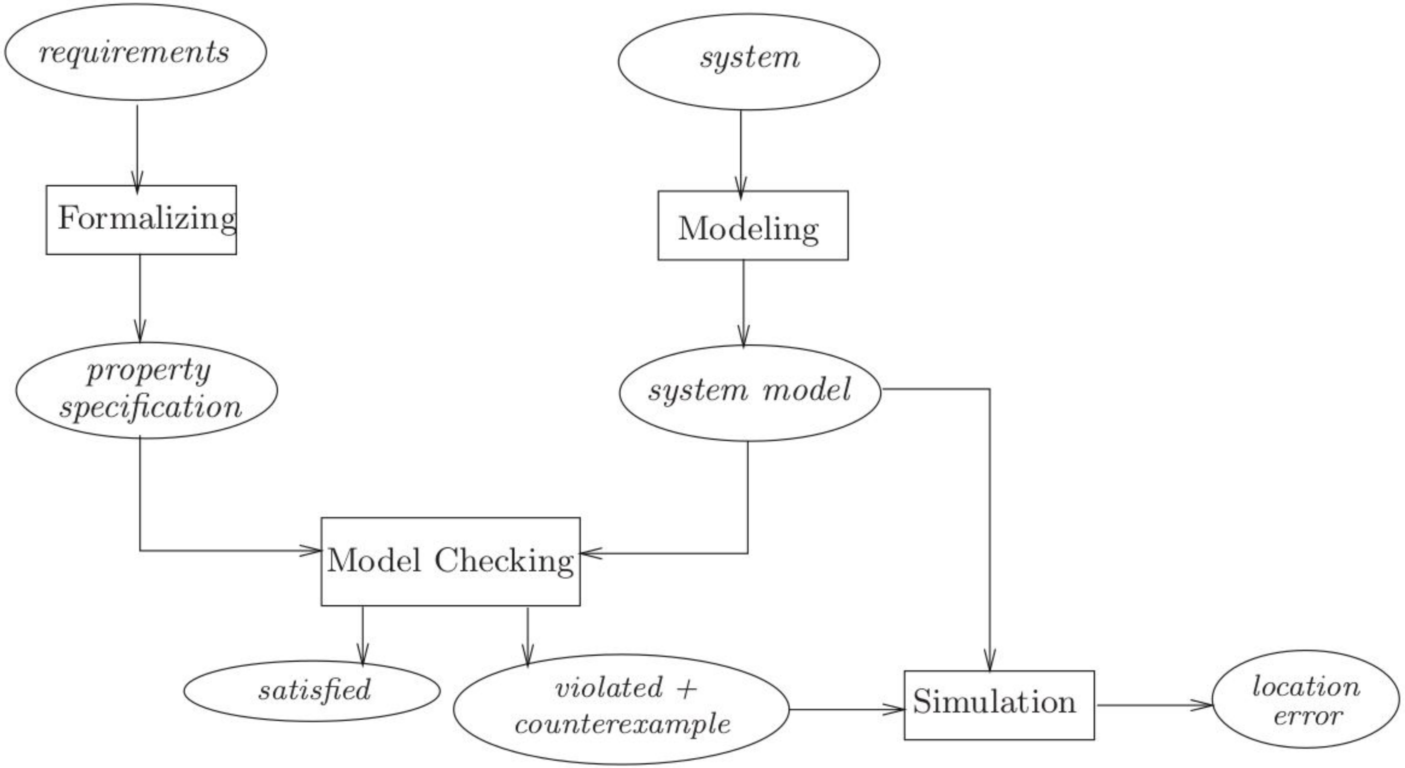
$$\Psi_1 \Psi_2 \ldots \Psi_n \vdash \Phi$$

$$\frac{\Psi \vdash \Phi_1 \wedge \Phi_2}{\Psi \vdash \Phi_1}$$

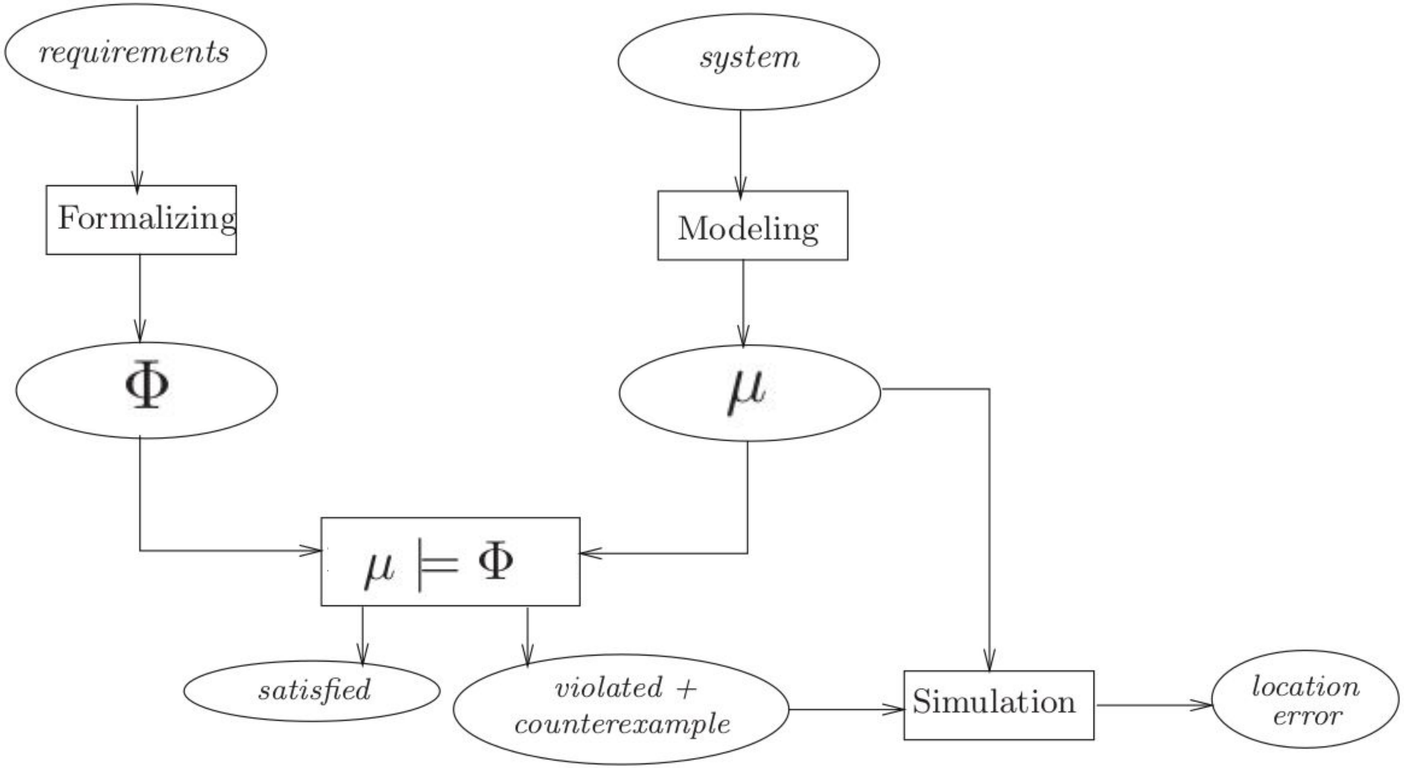
Model Checking from a Logical Perspective

1. Define a logic suitable for the properties of interest (additional operators w.r.t. classical propositional logic) Φ
2. Define a model-based semantics (with an appropriate mathematical entities in the role of the models) μ
3. (Define a translation from programming or modeling language to the set of chosen mathematical models)
4. Design a decision algorithm for $\mu \models \Phi$

Model Checking Approach Schema



Model Checking Approach Schema



An Example of Deductive Verification Method

Backward axiom

$$\frac{}{\{\mathcal{A}[e/x]\}\ x := e\ \{\mathcal{A}\}}$$

Invariant rule

$$\frac{\{\mathcal{I} \wedge b\}\ P\ \{\mathcal{I}\}}{\{\mathcal{I}\}\ \textbf{while}\ b\ \textbf{do}\ P\ \{\mathcal{I} \wedge \neg b\}}$$

Cut rule

$$\frac{\{\mathcal{A}\}\ P\ \{\mathcal{B}\} \qquad \{\mathcal{B}\}\ Q\ \{\mathcal{C}\}}{\{\mathcal{A}\}\ P; Q\ \{\mathcal{C}\}}$$

Logical rule

$$\frac{\mathcal{A} \Rightarrow \mathcal{A}' \quad \{\mathcal{A}'\}\ P\ \{\mathcal{B}'\} \quad \mathcal{B}' \Rightarrow \mathcal{B}}{\{\mathcal{A}\}\ P\ \{\mathcal{B}\}}$$

The strengths of model checking

- Widely applicable (hardware, software, protocols, configuration files, ...)
- Allows for partial verification (only most relevant properties)
- Not biased to the most possible scenarios (such as testing)
- Potential “push-button” technology (automated tools)
- Diagnostic information in case of property violation (counterexamples)
- Sound and interesting mathematical foundations (logics, graph algorithms …)

The weaknesses of model checking

- Decidability issues (check integer function termination?)
- Tractability issues (state explosion)
- No completeness for the logic (some property may be unexpressible)
- Main focus on control-intensive applications (less data-oriented)
- It is only as “good” as the system model
- It requires expertise in optimizing models and properties for efficiency
- It is not a compositional approach (verifying that two systems S_1 and S_2 satisfy a property P does not imply that their composition $S_1 \otimes S_2$ satisfies P)

Striking Model-Checking Examples

- **Security**: Needham-Schroeder public-key protocol: error that remained undiscovered for 17 years unrevealed
- **Transportation systems**: train model containing 10476 states
- **Programming Languages**: Model checkers for C, Java and C++ used (and developed) by Microsoft, NASA, etc. (device drivers)
- **Hardware Verification**: Successful applications of (symbolic) model checking to large hardware systems, part of the hardware development process at IBM
- **Space**: Formal analysis of Mars Science Laboratory, Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc.
- **Health**: Verification of medical device transmission protocols

An Important Field of Application: Concurrent Programs

Consider these three threads and assume x = 0

```
while true do
    if x < 200 then
        x := x + 1
od
```

```
while true do
    if x > 0 then
        x := x - 1
od
```

```
while true do
    if x = 200 then
        x := 0
od
```

Verify: is x always between (and including) 0 and 200?

```
1. while true do
2.     if x < 200 then
3.             x := x + 1
4. od
```

```
1. while true do
2.     if x > 0 then
3.             x := x - 1
4. od
```

```
1. while true do
2.     if x = 200 then
3.             x := 0
4. od
```

(**x = 0**, pc1 = 2, pc2 = 2, pc3 = 2)

↓

(**x = 0**, pc1 = 3, pc2 = 2, pc3 = 2)

↓

(**x = 1**, pc1 = 1, pc2 = 2, pc3 = 2)

↓

(**x = 1**, pc1 = 2, pc2 = 2, pc3 = 2)

↓

(**x = 1**, pc1 = 3, pc2 = 2, pc3 = 2)

↓

(**x = 2**, pc1 = 1, pc2 = 2, pc3 = 2)

↓

(**x = 200**, pc1 = 1, pc2 = 2, pc3 = 2)

↓

(**x = 200**, pc1 = 1, pc2 = 3, pc3 = 2)

↓

(**x = 200**, pc1 = 1, pc2 = 3, pc3 = 3)

↓

(**x = 0**, pc1 = 1, pc2 = 3, pc3 = 1)

↓

(**x = -1**, pc1 = 1, pc2 = 1, pc3 = 1)

Using Spin Model Checker

```
int x = 0;

proctype Inc() {
  do :: true -> if :: (x < 200) -> x = x + 1 fi od
}

proctype Dec() {
  do :: true -> if :: (x > 0) -> x = x - 1 fi od
}

proctype Reset() {
  do :: true ->  if :: (x == 200) -> x = 0 fi od
}
```

```
proctype Check() {
  assert (x >= 0 && x <= 200)
}

init {
  atomic{ run Inc() ; run Dec() ; run Reset() ; run Check() }
}
```

```
spin: text of failed assertion: assert(((x>=0)&&(x<=200)))
```

We can fix the problem by imposing atomicity

Model Checking Course Syllabus

- **Modelling Systems**
  - Transition systems and program graphs
  - Modelling Concurrent Systems
- **Linear Time Properties**
  - Invariants, Safety, Liveness and Fairness
  - Checking regular safety properties
  - Checking omega regular properties with Büchi automata
- **Linear Time Logics**
  - Positive Normal Forms
  - Fairness
  - Model checking LTL formulas
- **Branching Time Logics**
  - Computational Tree Logics
  - Comparison of the expressivity of LTL, CTL and CTL*
  - Model checking CTL and CTL* formulas