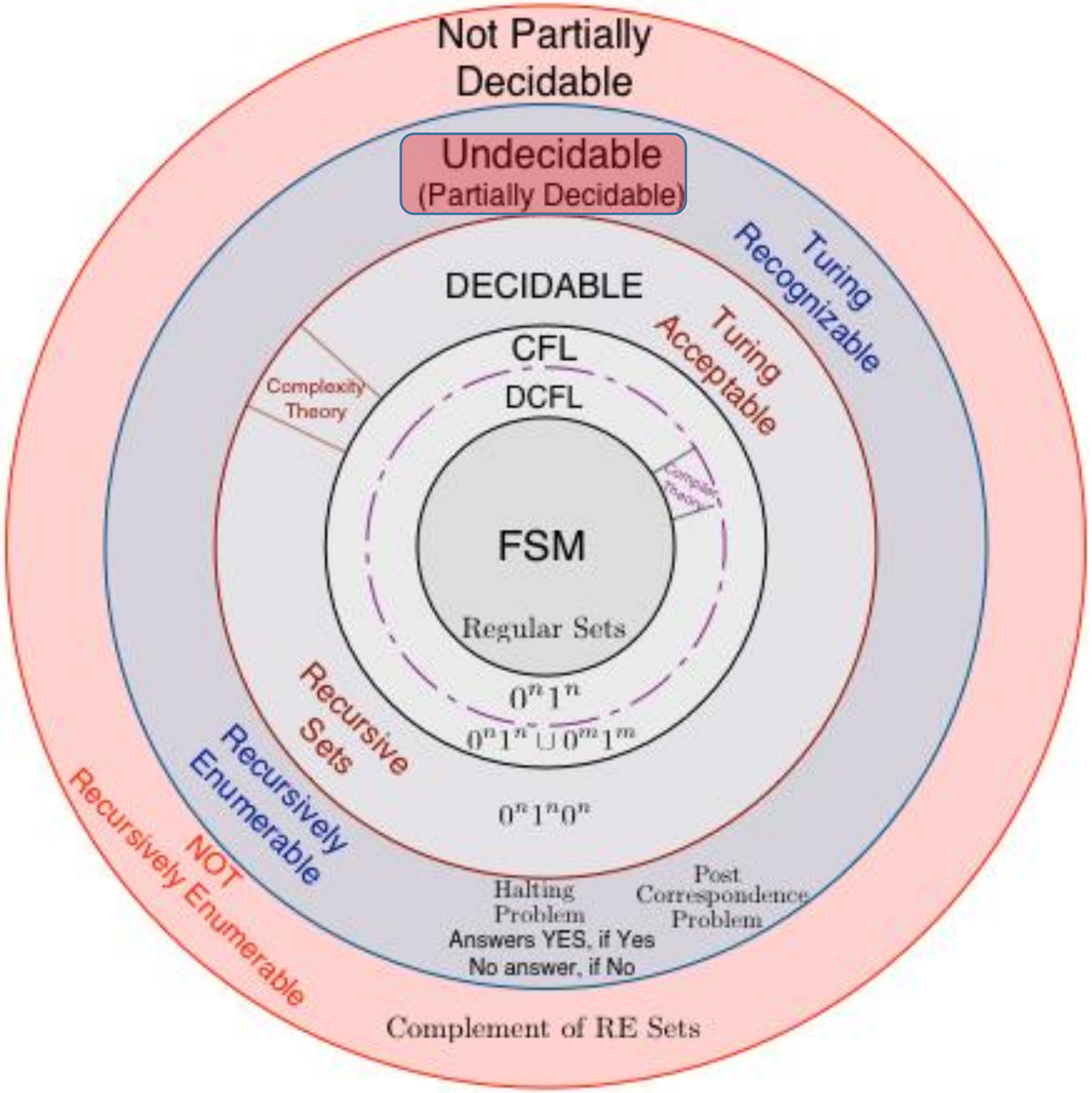
Theoretical Computer Science

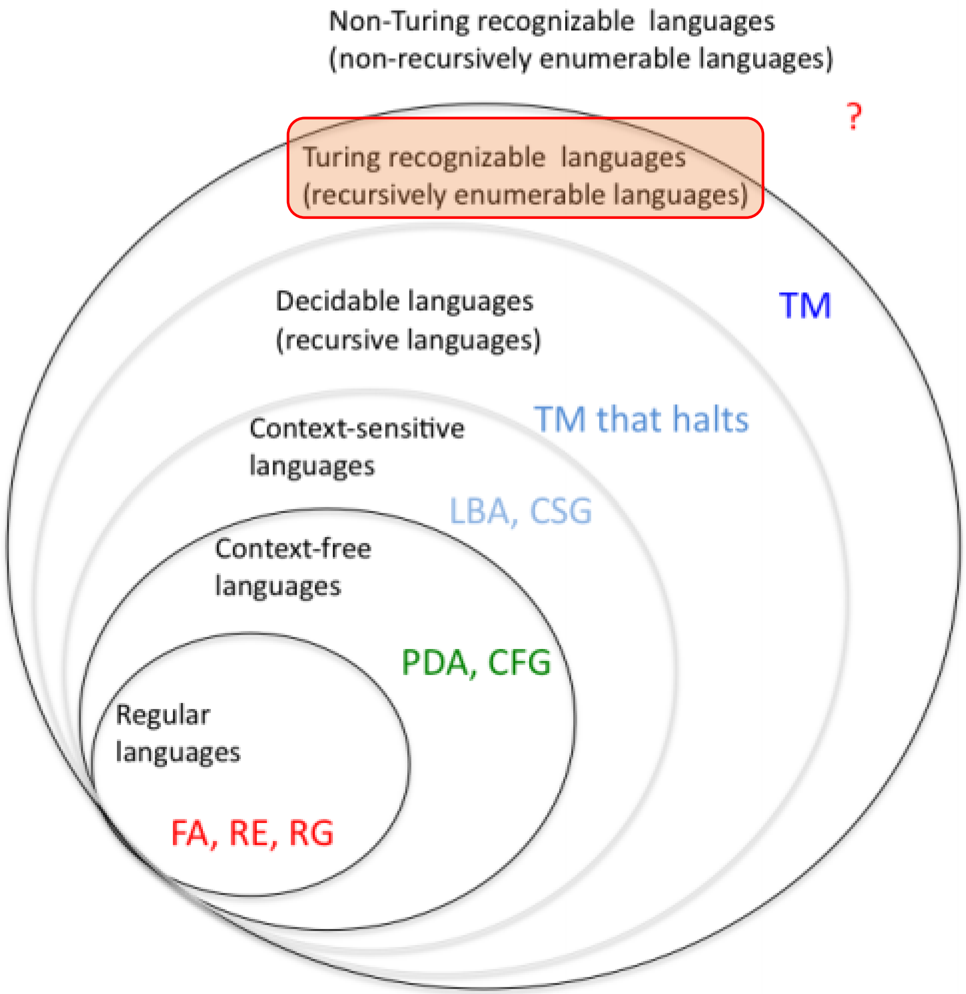
More on Computability Theory

Lecture 15 - Manuel Mazzara


Not Partially Decidable
Undecidable
(Partially Decidable)
Turing Recognizable
Turing Acceptable
DECIDABLE
CFL
DCFL
Complexity Theory
FSM
Regular Sets
$0^n 1^n$
$0^n 1^n \cup 0^m 1^m$
$0^n 1^n 0^n$
Recursive Sets
Recursively Enumerable
NOT Recursively Enumerable
Halting Problem
Answers YES, if Yes
No answer, if No
Post Correspondence Problem
Complement of RE Sets

Recursively enumerable sets

Recursively enumerable sets in context



FA: finite state automaton
RE: regular expression
RG: regular grammar

CFG: context-free grammar
PDA: pushdown automaton

LBA: linear-bounded automaton
CSG: context-sensitive grammar
TM: Turing machine

Theorem

- Consider the set S with the following features:
  - $i \in S \rightarrow \mathbf{f_i}$ total (i.e., **S contains only indexes of total computable functions**)
  - f total and computable $\rightarrow \exists\, i \in S \mid \mathbf{f_i} = f$ (i.e., **S contains all of them**)
  - **S is the set of total computable functions**
  - **S is not RE**
  - Provable by diagonalization (homework)

Implications (1)

- **There is no RE formalism (Automata, grammars, TMs …) that can define all computable total functions, and only them**

- FSA define total computable functions, but <u>not all of them</u>
  - Model with predetermined fixed memory is less powerful than typical programming languages

- TMs define all computable functions, but <u>including also the non-total ones</u>
  - **Non termination as a features of programming languages**

Implications (2)

- C programming language allows coding any algorithm, including the non-terminating ones (Turing-powerful)
- **There is no subset of C that defines exactly all and only the terminating programs**
- The set of C programs in which **loops comply with given constraints guaranteeing termination** includes terminating programs only, but necessarily **not all terminating programs**

There is no RE formalism (Automata, grammars, TMs …) that can define all computable total functions, and only them

Let us climb upper!

Do you know that

$\cos(0.7390851332) = 0.7390851332$?

In mathematics, a **fixed point** of a function f is a value x such that $f(x) = x$

Kleene’s fixed-point theorem

- Let **t** be a total and computable function. Then it is always possible to find an integer ***p*** such that $\mathbf{f_p = f_{t(p)}}$
  - Function f_p is called a **fixed point** of t

- Kleene’s Fixed point theorem (1938)
- Proof as homework
- We will use here to prove the Rice’s theorem
- **For any total computable function f, there is a number p such that both *p* and *t(p)* indicate the same computable function**

Theoretical Computer Science

Rice's theorem

Lecture 15 - Manuel Mazzara

Rice’s Theorem

Henry Gordon Rice, 1951

Rice’s theorem, formally

Let **F** be a set of computable functions. We define the set **S** of (the indices of) TMs that compute the functions of **F**:

$$\mathbf{S = \{ x \mid f_x \in F \}}$$

S is **decidable if and only if F = ∅ or F is the set of <u>all computable functions</u>**

<u>In all nontrivial cases S is not decidable</u>

Rice's theorem informally

A property that holds for every machine or holds for none is trivial

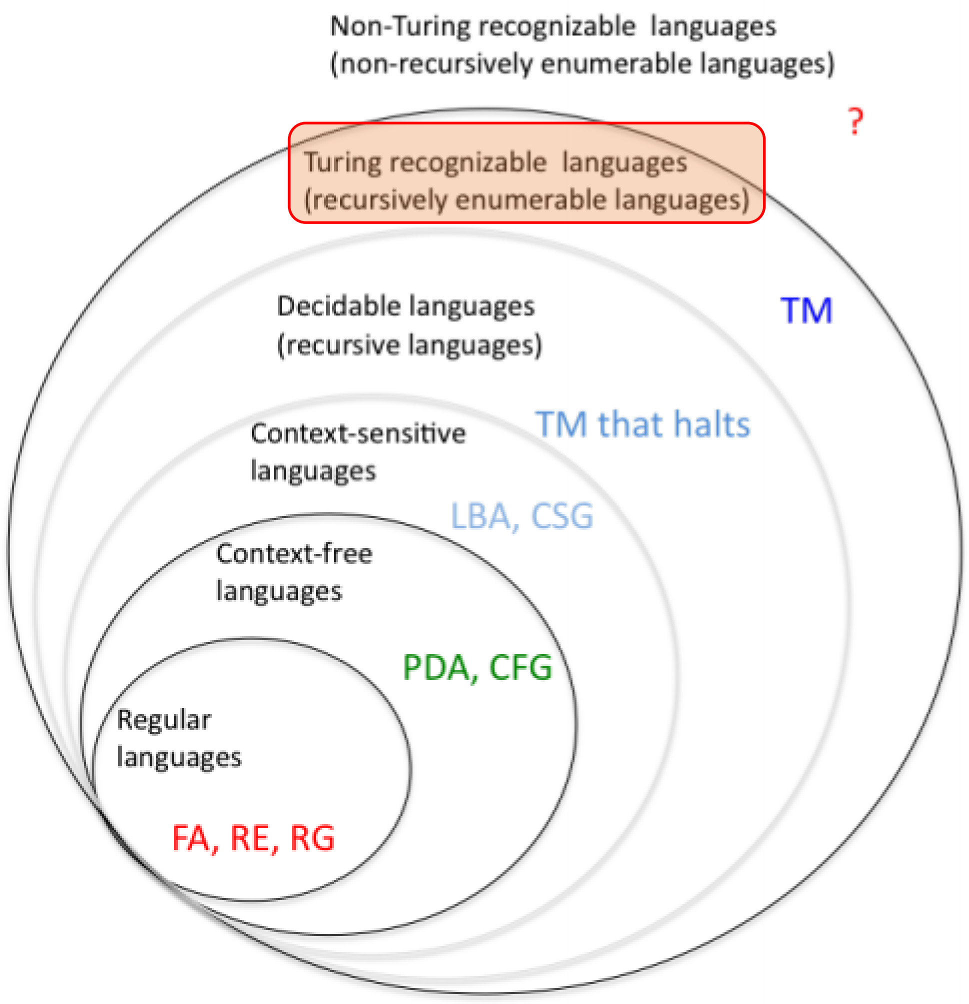
Every <u>nontrivial</u> property of the recursively enumerable languages is undecidable

Let us try to formulate it in programming terms

For all (non-trivial), **semantic** properties of programs it is impossible to construct an algorithm that always leads to **a correct yes-or-no answer** to the question on whether the program satisfies the property or not

Semantic means regarding the *behavior* of programs. Non-trivial means that the property is true for *some* program, but not for all or none

Recursively enumerable sets in context



FA: finite state automaton
RE: regular expression
RG: regular grammar

CFG: context-free grammar
PDA: pushdown automaton

LBA: linear-bounded automaton
CSG: context-sensitive grammar
TM: Turing machine

Practical implications (examples)

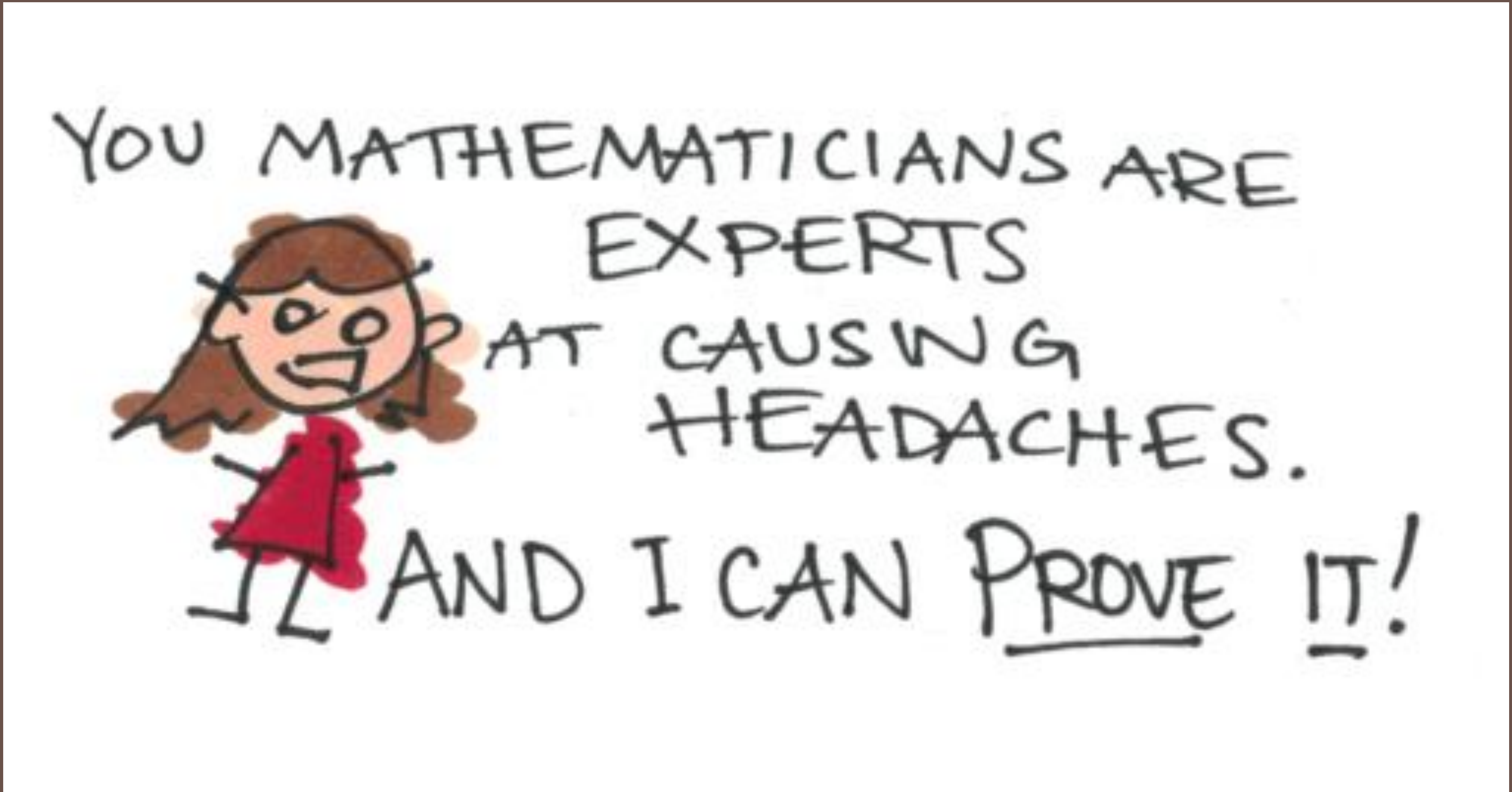
- Program correctness: **does P solve a given specified problem?**
  - Does M_x compute a function that is in the set {f}?

- **Program equivalence**
  - Does M_x compute the function that constitutes the singleton set $\{f_y\}$?

- **Does a program have any property concerning the function it computes?**
  - Function with even values, function with a limited image, …

If the property under analysis is nontrivial (meaning it does not belong to all or no program), then the corresponding computational decision problem will not be decidable (it may be semi-decidable though, see HP)

Let us see the formal proof…


YOU MATHEMATICIANS ARE EXPERTS AT CAUSING HEADACHES.
AND I CAN PROVE IT!

Proof (1)

- Suppose that:
  - S is recursive,
  - $F \neq \emptyset$ and
  - F is not the set of all computable functions
- Let us consider the characteristic function c_S of **S**
  - **$c_S(x)$ =if $f_x \in$ F then 1 else 0**
- By hypothesis, c_S is total and computable, so by enumerating each TM M_i, we can find

  (1) the first $i \in S$ such that $f_i \in F$ and

  (2) the first $j \notin S$ such that $f_j \notin F$

Proof (2)

- Since c_S is computable, then so is c'_S:

$$\mathbf{c'_S(x)= if\ f_x \notin F\ then\ i\ else\ j} \qquad (3)$$

- By **Kleene's theorem**, there exists an x' such that

$$f_{c'_S(x')}=f_{x'} \qquad (4)$$

- Let us consider $\mathbf{c'_S(x')}$. There are two cases:
  - **Suppose $c'_S(x')=i$** then by (3) $\mathbf{f_{x'} \notin F}$, but by (4) $\mathbf{f_{x'} = f_i}$ and by (1) $\mathbf{f_i \in F}$: **contradiction, we have both $f_{x'} \notin F$ and $f_{x'} \in F$**
  - **Suppose instead $c'_S(x')=j$**, then by (3) $\mathbf{f_{x'} \in F}$, but by (4) $\mathbf{f_{x'}=f_j}$ and by (2) $\mathbf{f_j \notin F}$: **contradiction, we have both $f_{x'} \in F$ and $f_{x'} \notin F$**

Consequences

- Rice's theorem has strong **negative implications**:
  - There is an endless list of interesting problems whose undecidability follows trivially from Rice's theorem

- For any chosen set F={*g*}, by Rice's theorem it is not decidable whether a generic given TM computes *g* or not

Syntactic vs. Semantic properties

- A **syntactic property** is purely about program structure
  - Does the program contain an if-then-else statement?
- A **semantic property** is about program's behaviour
  - Does the program terminate for all inputs?
- A property is **non-trivial** if it is neither true for every program, nor for no program
- **Rice's theorem** states that all non-trivial, semantic properties of programs are undecidable

Summary

- For every **non-trivial property of partial functions**, no general and effective method can decide **whether an algorithm computes a partial function with that property**

- **Any interesting property of program behavior is undecidable**

Any interesting property of program behavior is undecidable

What to do?

It is impossible to fully automatize software verification

Software verification is about **engineering workaround to the fundamental problems**

Approximate solutions exist and we can still live our life!

Theoretical Computer Science

Final Summary

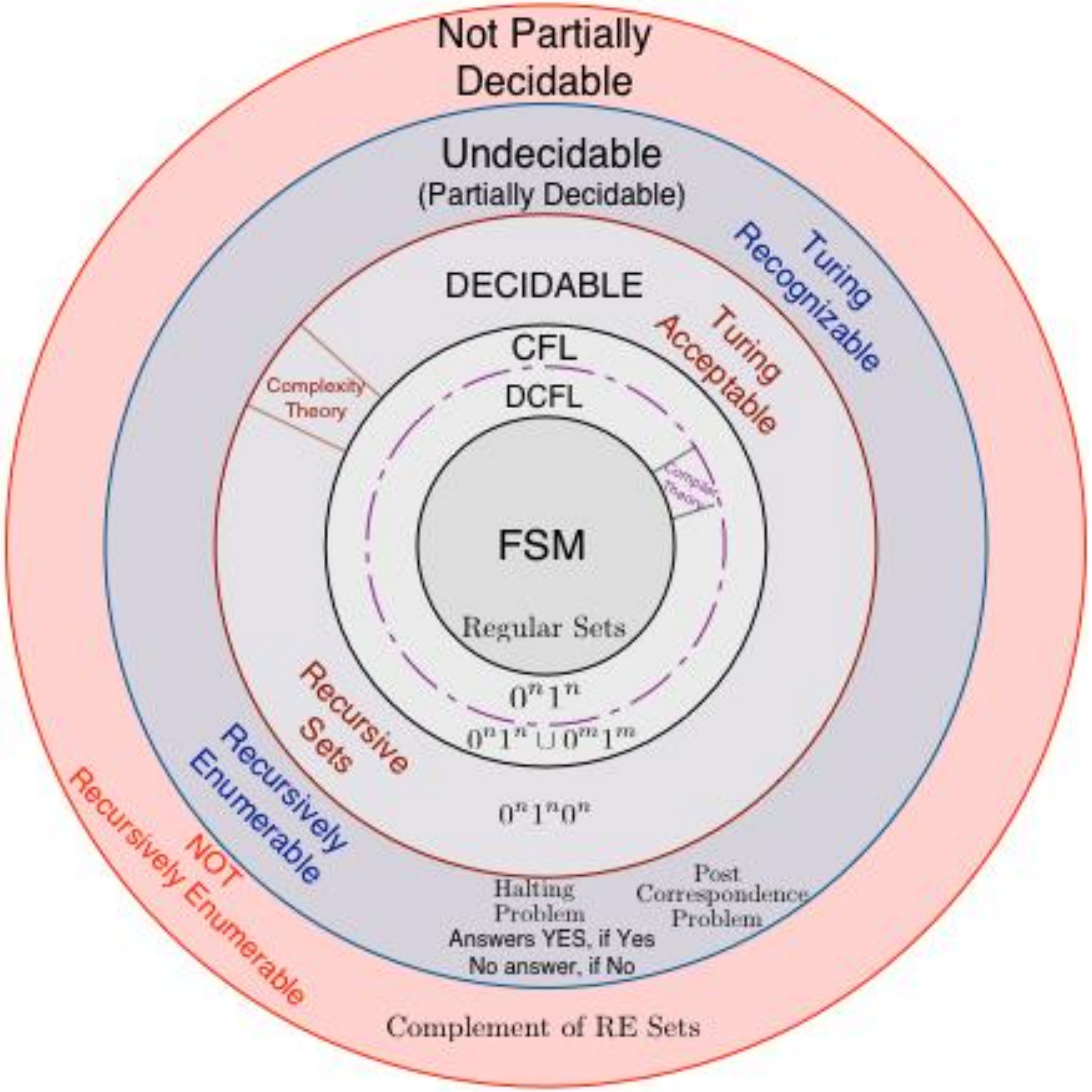
Lecture 15 - Manuel Mazzara

A few slides that summarize pretty much everything…

From regular sets to recursively enumerable

- Inner-outer: **more and more expressive automata and grammars**
- Strict inclusion
- **Different property of closure** (**<u>complement</u>**…)
- **Property of programming languages** (**<u>RE languages</u>**)

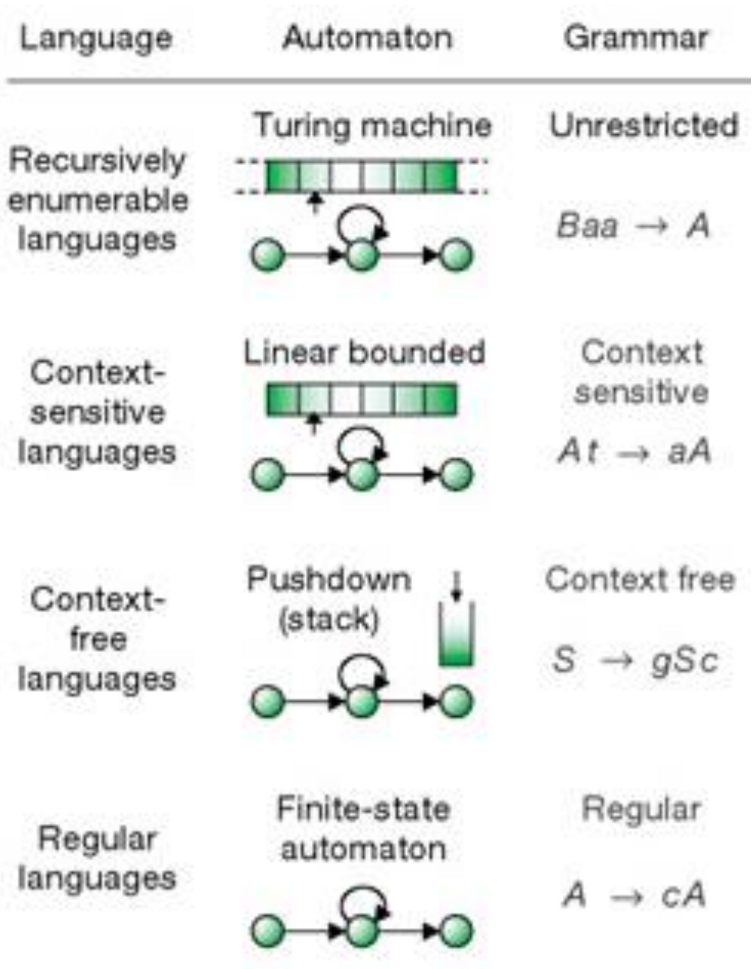
- **<u>Open question</u>**: how would you comment the following diagram? What kind of conclusions/considerations can you draw about it?


Not Partially Decidable
Undecidable
(Partially Decidable)
Turing Recognizable
DECIDABLE
Turing Acceptable
CFL
DCFL
Complexity Theory
FSM
Regular Sets
$0^n 1^n$
$0^n 1^n \cup 0^m 1^m$
Recursive Sets
Recursively Enumerable
$0^n 1^n 0^n$
NOT Recursively Enumerable
Halting Problem
Answers YES, if Yes
No answer, if No
Post Correspondence Problem
Complement of RE Sets

Correspondence automata-grammars

- **From least expressive to most expressive**
- **Restrictions**
  - On **memory model**
  - On **productions shape**
- Different kind of **memory model** and **production**

- **Open question**: How would you describe the different memory models and the corresponding rules on productions?



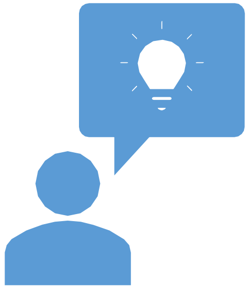
Productions have no restrictions

Rewrite a nonterminal according to the context

Context does not count

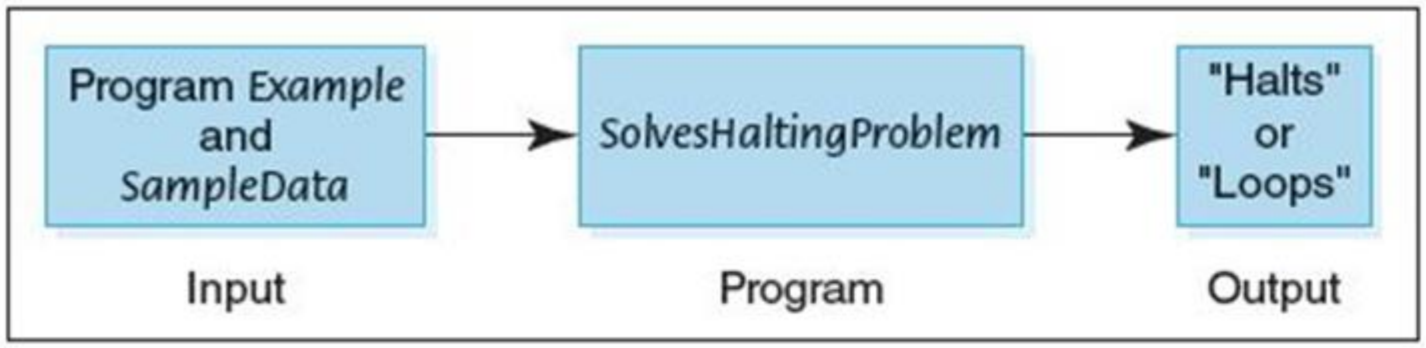
Rewrite a nonterminal as a terminal
followed by at most one nonterminal

Why do we need two different “worlds”, one **operational** and one **generative**?



Halting Problem

- Given a **program** and an **input to the program**, determine if the given program **will eventually stop** with this particular input



Why HP is relevant?
What does it tell us?

Exercise

- At the end of this last class:
  - Go through these recap questions
  - Build a narrative connecting all these questions and all the topics of the course

- **Are you able to see the bigger picture now?**

Continuation of TOC

- **Complexity Theory**
  - You should have seen in other courses
- **Compilers**
  - We have seen some hints
  - You will study compilers construction in SE track
- **Software Formal Verification**
  - Elective course or in MSIT program

Further readings

- **The concept of computability**, Carol E. Cleland, *Theoretical Computer Science*, Volume 317, Issues 1–3, 2004
- **Computability Theory,** Wilfried Sieg, *Philosophy of Mathematics*, edited by Andrew D. Irvine, Elsevier, 2009

Theoretical Computer Science

Software Formal verification (hints)

Lecture 15 - Manuel Mazzara

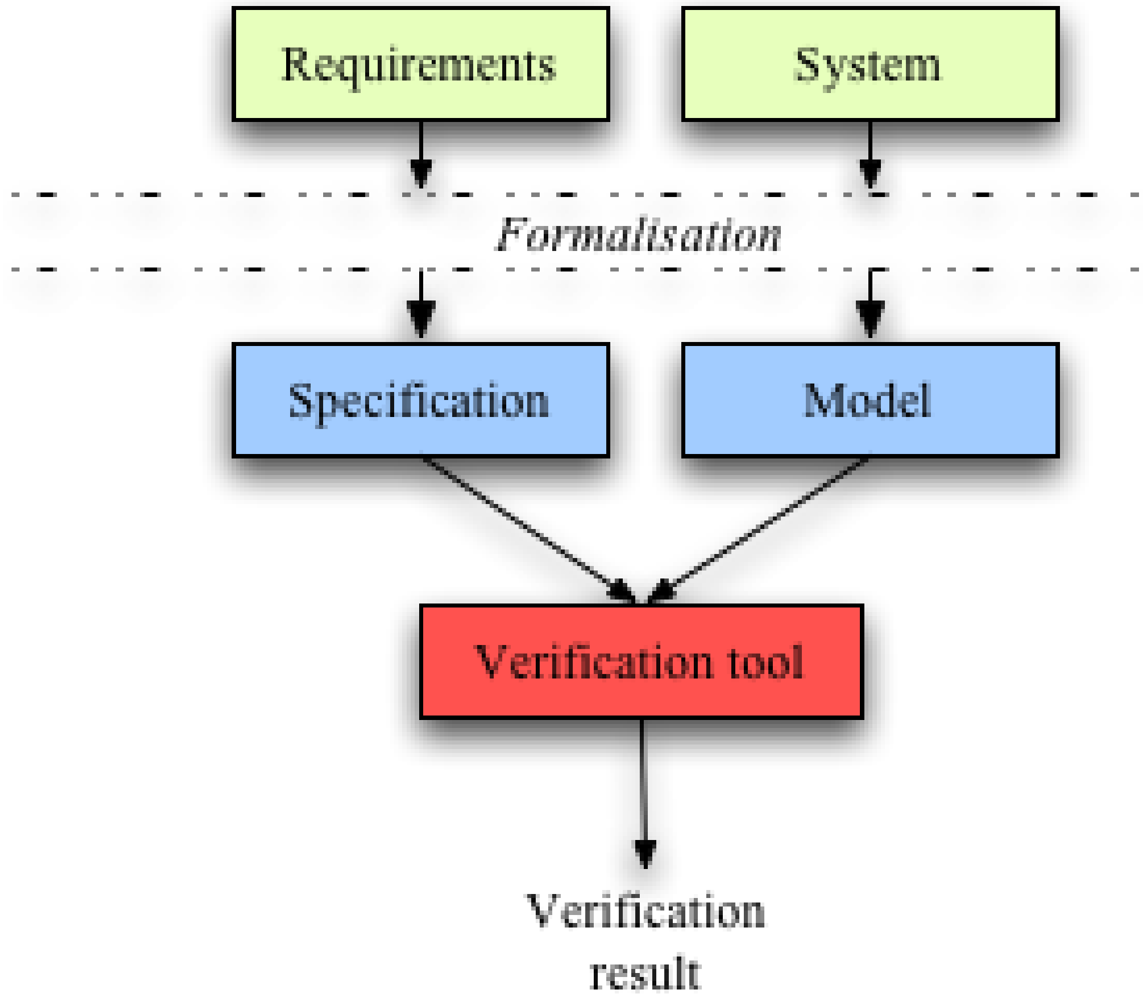
Formal verification

- **Formal verification** means using methods of mathematical argument to determine **correctness of systems**
  - Can be applied to hardware and software

- Bugs are expensive when discovered in a finished product
  - Idea: **use Formal Verification (FV) to discover bugs during the *design* phase**

What do we need?

- It is necessary to describe:
  - A **model of the system** to be verified
  - A **specification of the properties** to be checked



Program Verification: the idea

- The **Program Verification problem**
- Given: a program **P** and a specification **S**
- Determine: if **every execution of P, for any value of input arguments, satisfies S**

Remember!

- For every **non-trivial property of partial functions**, no general and effective method can decide **whether an algorithm computes a partial function with that property**

- **Any interesting property of program behavior is undecidable**

Program Verification: limits (1)

- The very nature of universal (Turing-complete) computation entails the **impossibility of deciding automatically the program verification problem**

P: a program

$\Updownarrow$

TM(P): a Turing machine

S: a specification

$\Updownarrow$

F(S): a first-order formula

Program Verification: limits (2)

Does $TM(P) \vDash F(S)$ hold?

UNDECIDABLE

What can be done?

- Restricting the expressiveness of:
  - the computational model
  - the specification language

- **The verification problem may become decidable**

Program Verification and Model Checking

- **The Program Verification problem is decidable if P is finite-state**

- Real programs are not finite-state
  - arbitrarily complex inputs
  - dynamic memory allocation
  - …
- The term Software Model-Checking denotes techniques to **automatically verify real programs based on finite-state models of them**
  - It is a convergence of verification techniques developed during the late 1990's

Model Checking

- Model checking is a technique for verifying **finite state concurrent systems**

- Advantages over other traditional approaches:
  - Model checking is **automatic**
  - If the design contains an error, model checking will produce **a counterexample** that can be used to pinpoint the source of the error

- Challenge:
  - dealing with the **state space explosion problem**

The idea

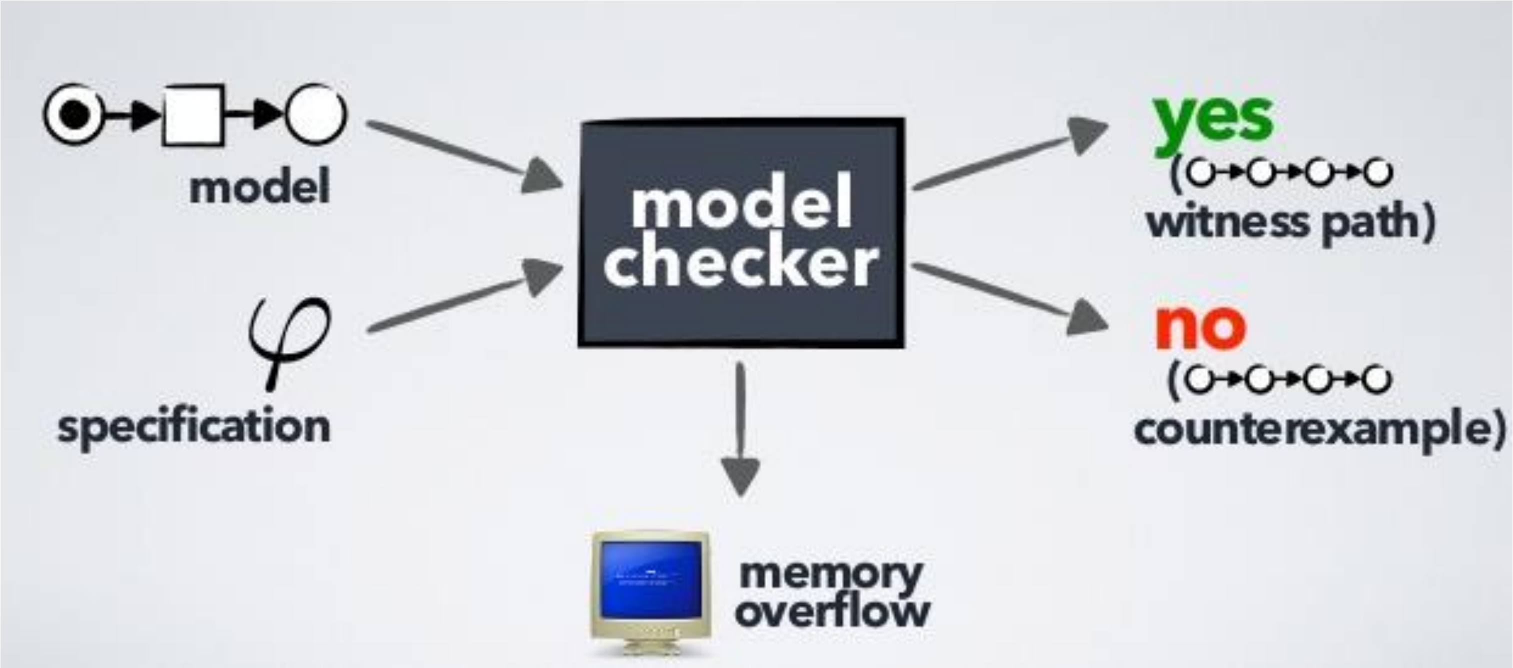
- The idea is dramatically simple in its fundamentals

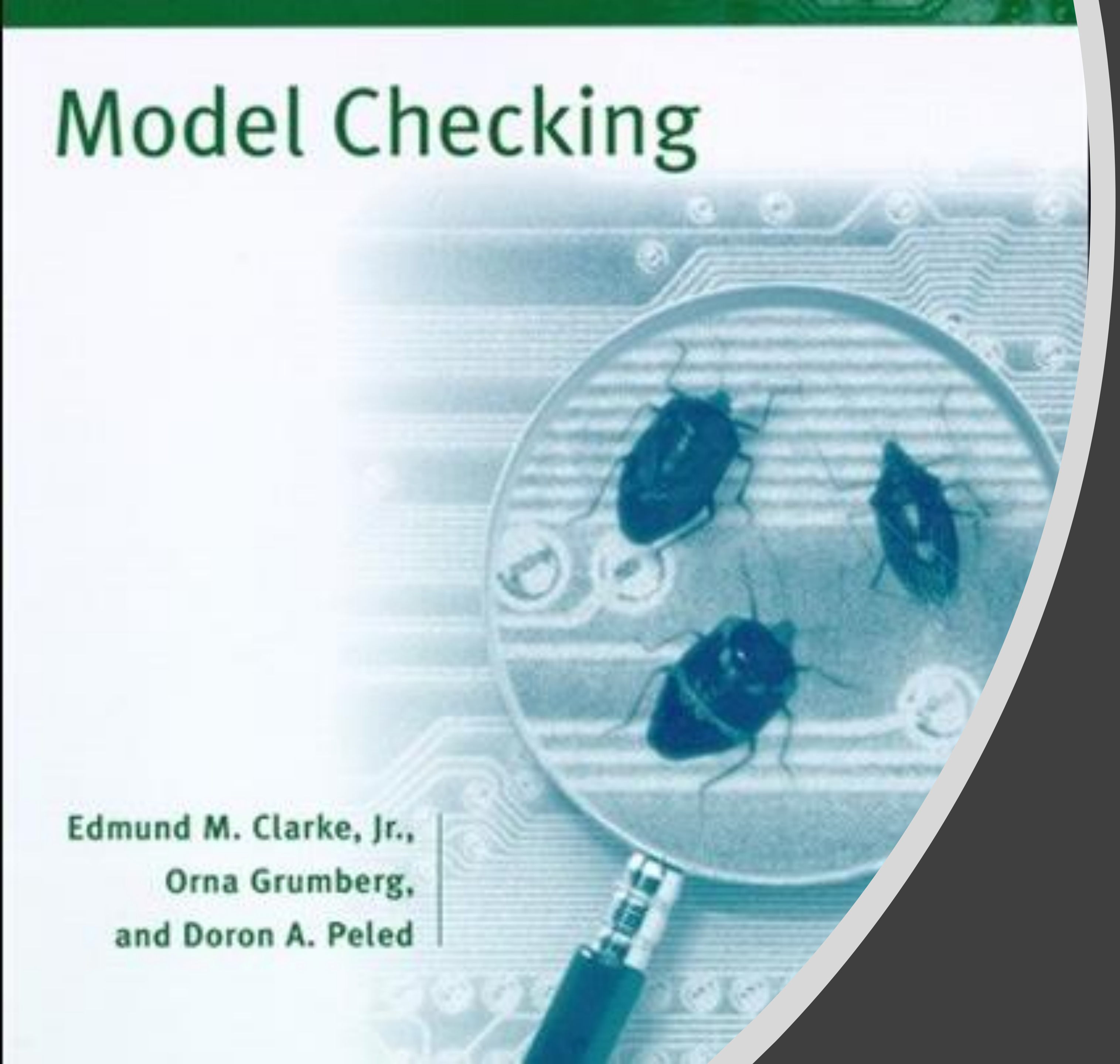
- Specifications are formulas *f* in **propositional temporal logic**
  - Extension of propositional logic with operators to describe properties of **dynamic systems** (truths changing over time)

- **Models are specific kind of Finite-State Automata (Kripke structure)**
  - With state labelling and other small differences

Verification procedure: **exhaustive** (but efficient) **search of the state space** of the model to see if it satisfies f

Model Checking Process





Suggested reading – if interested

Thank you for coming and attending until the very end!


"Always treat people as ends in themselves, never as means to an end."
– Immanuel Kant, Groundwork of the Metaphysics of Morals, 1785