Software Verification

# Introduction
# Model Checking and Temporal Logic

Ahmed Rezine

IDA, Linköpings Universitet

Vårtermin 2026

# Outline

# Outline

# This course

- Introduces principals behind software verification approaches including model checking, Hoare-style reasoning, satisfiability modulo theory and abstract interpretation
- Uses **assignments** to allow for experimenting with the introduced notions. The assignments will have a "theoretical" part and a "practical" part. The practical part involves hands-on assignments on representative tools of the different verification techniques.
- Concludes with an **exam**
- Course homepage and updates

# Plan

- Explicit model checking
  - 2 lectures + Assignment.
- Axiomatic reasoning
  - 2 lectures + Assignment.
- Bounded/symbolic verification
  - 2 lectures + Assignment.
- Scalable over-approximation
  - 2 lectures + Assignment.
- Exam

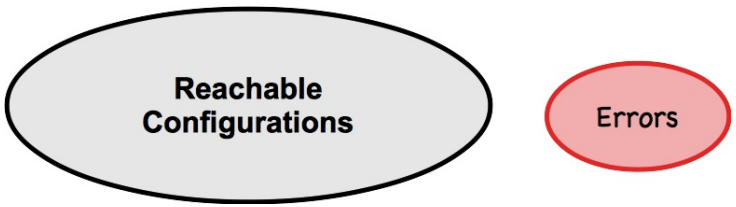
# Assignments

- There are four assignments
- Expected to work in pairs
- Register to webreg before April 8th
- Demonstrate your solution in a scheduled lab session
- Deadline for demonstration: last lab session
- Deadline for submission: one week after last lab session

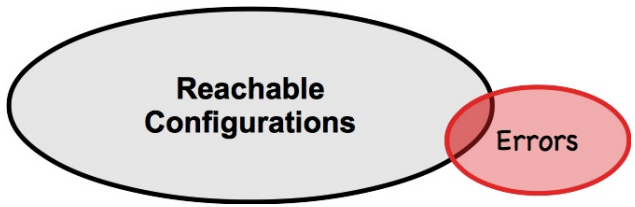
# Outline

# Verification

- We want to answer whether some program behaves correctly. We define "correctness" soon.
- For now, assume that means some erroneous configurations are not reachable
- We say the program is **safe**



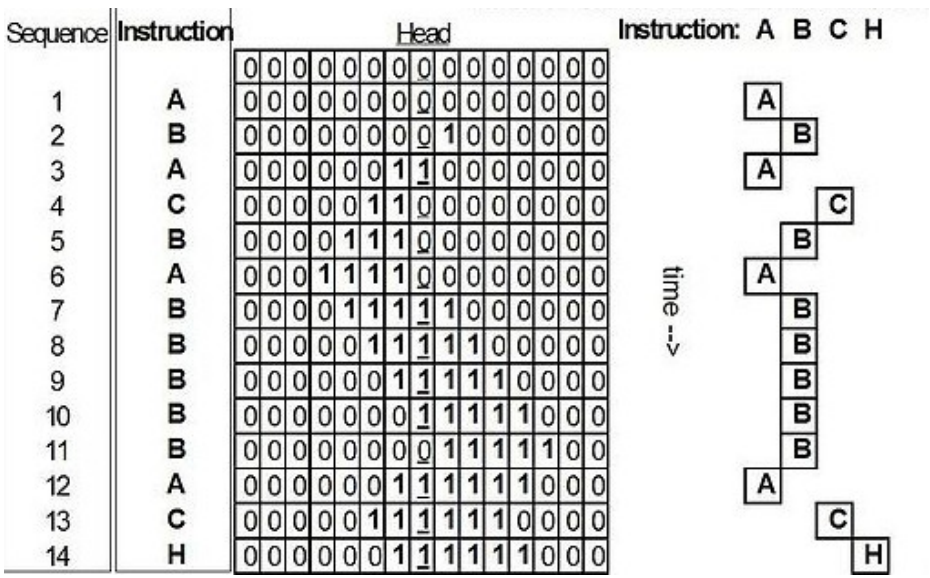

Safe Program

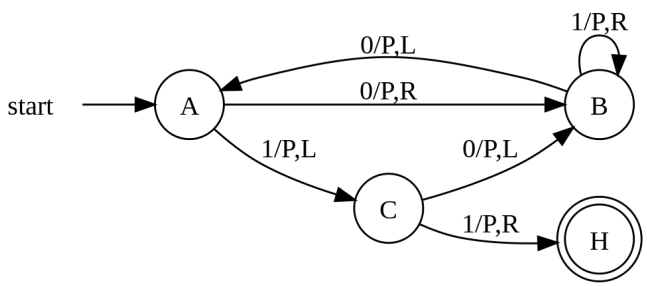



Unsafe Program

# The general verification problem is “very difficult”

- Deciding whether all possible executions of a program are error-free is hard. If we could write a program that does it for arbitrary programs to be analyzed then we would always be able to answer whether a Turing machine halts.
- This problem is proven to be undecidable.



# Problem is “very difficult”: what to do?

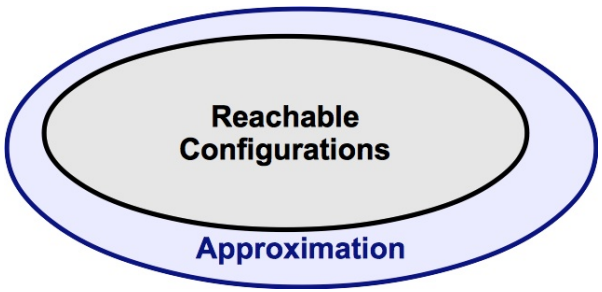
- Identify sub-problems on which one can decide: e.g. finite state machines, push-down automata, timed automata, Petri nets, well-structured transition systems.
- Proceed with approximations that will hopefully give some guarantees.

# Verification problem and approximations

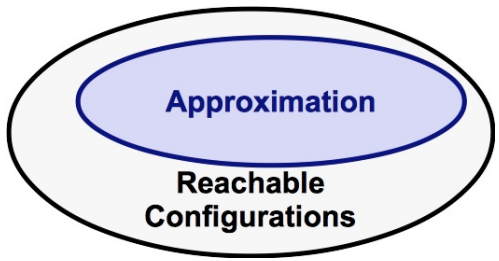
- An analysis procedure takes as input a program to be checked against a property. The procedure is an analysis algorithm if it is guaranteed to terminate.
- An analysis algorithm is **sound** in the case where each time it reports the program is safe wrt. some errors, then the original program is indeed safe wrt. those errors (pessimistic analysis)
- An algorithm is **complete** in the case where each time it is given a program that is safe wrt. some errors, then it does report it to be safe wrt. those errors (optimistic analysis)
- In general, you have to give up on one of the three: termination, soundness or completeness.

# Verification problem and approximations

- The idea is then to come up with efficient approximations to give correct answers in as many cases as possible.



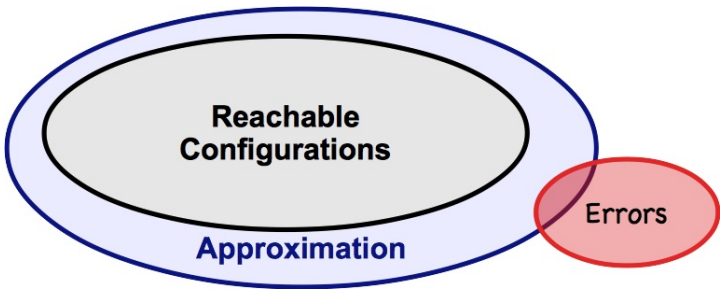

Over-approximation



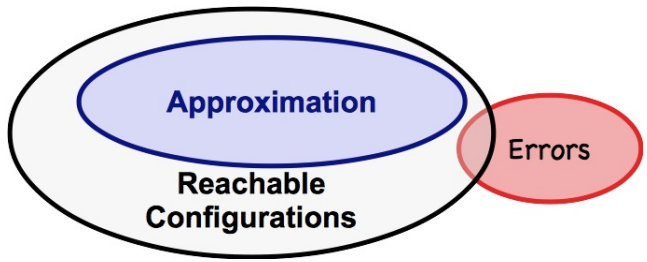

Under-approximation

# Program verification and the price of approximations

- A sound analysis cannot give **false negatives**
- A complete analysis cannot give **false positives**




False Positive




False Negative

# We will introduce the following techniques

- Explicit model checking
  - represents “behaviors” explicitely. Aims for exactness on sub-classes.
- Symbolic based techniques
  - represents “behaviors” symbolically. Can be used for sound or complete approaches.
- Axiomatic reasoning
  - Uses predicates and can prove anything provable by a human, but with human intervention.
- Scalable over-approximation
  - Uses sound approximations that may lead to false positives.

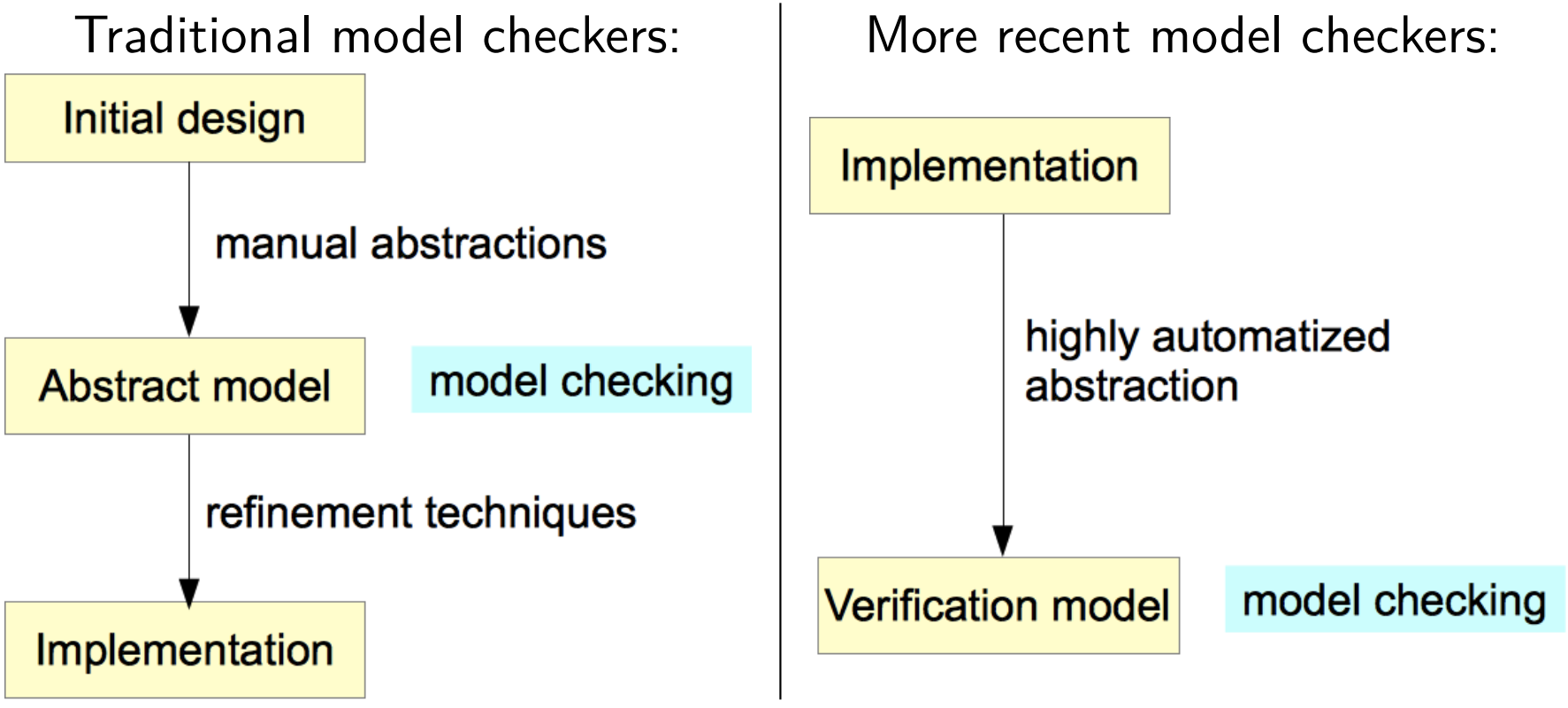
# Outline

# Model checking

$$M \stackrel{?}{\models} \Phi$$

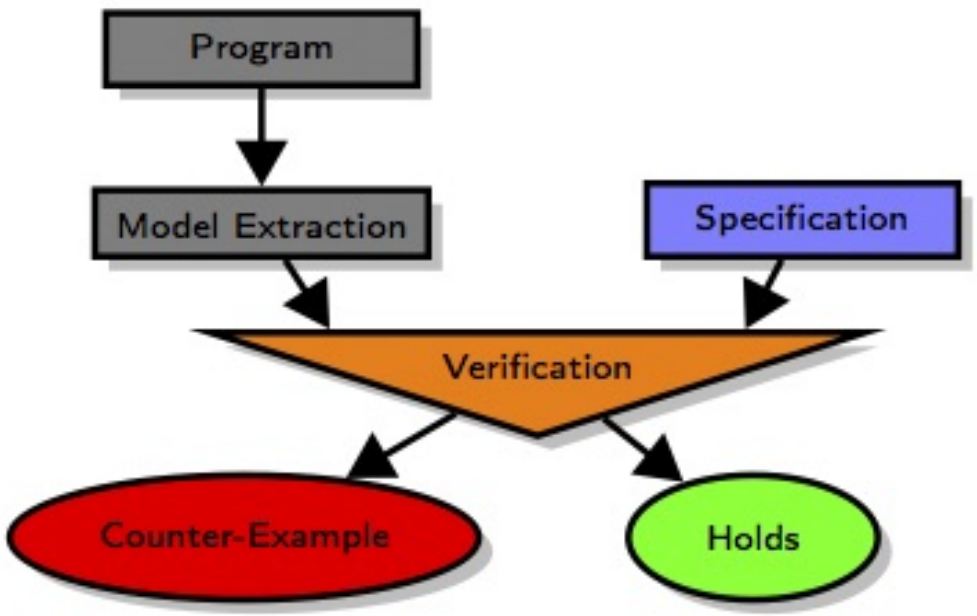
- Model checking is a push button verification approach
- Given:
  - a model $M$ of the system to be verified, and
  - a correctness property $\Phi$ to be checked: absence of deadlocks, livelocks, starvation, violations of constraints/assertions, etc
- The model checking tool returns:
  - a counter example in case $M$ does not model $\Phi$, or
  - a "proof" that $M$ does model $\Phi$

# Model Checking in Practice



# Model Checking: Verification vs debugging

- Model checking tools are used both:
  - To establish correctness of a model $M$ with respect to a correctness property $\Phi$
  - More importantly, to find bugs and errors in $M$ early during the design



# $M$ as a Kripke structure

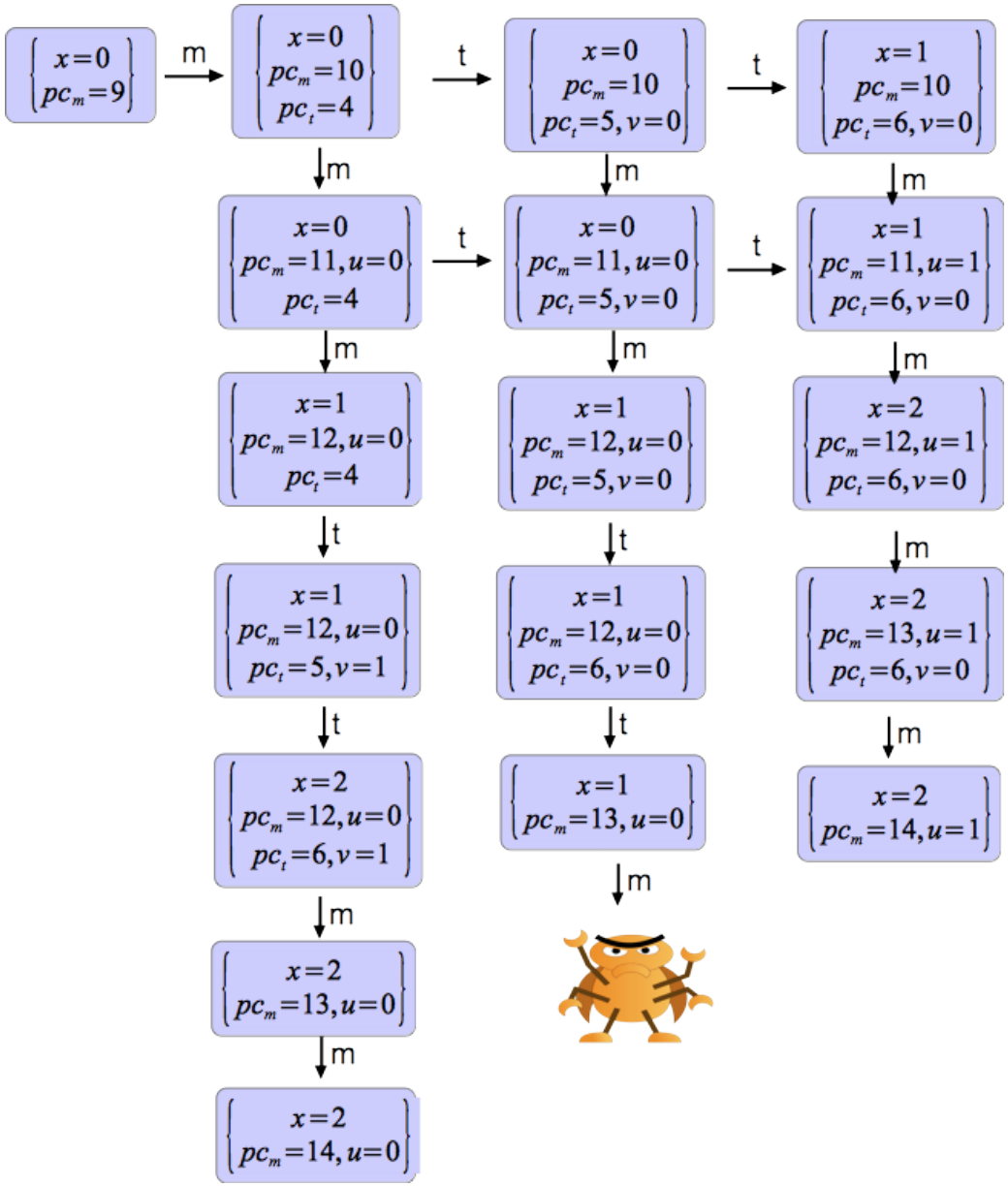
Assume a set of atomic propositions $AP$. A *Kripke structure* $M$ is a tuple $(S, S_0, R, L)$ where:

1. $S$ is a finite set of states
2. $S_0 \subseteq S$ is the set of initial states
3. $R \subseteq S \times S$ is the transition relation s.t. for any $s \in S$, $R(s, s')$ holds for some $s' \in S$
4. $L : S \to 2^{AP}$ labels each state with the atomic propositions that hold on it.

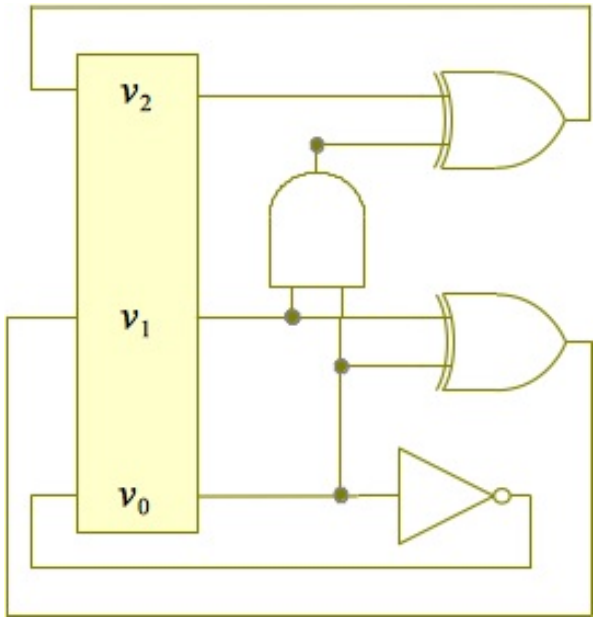
Intuitively, $AP$ are properties whose evolution (when moving from one state to the other) we want to track. Kripke structures can be used to capture the behavior of very different systems.

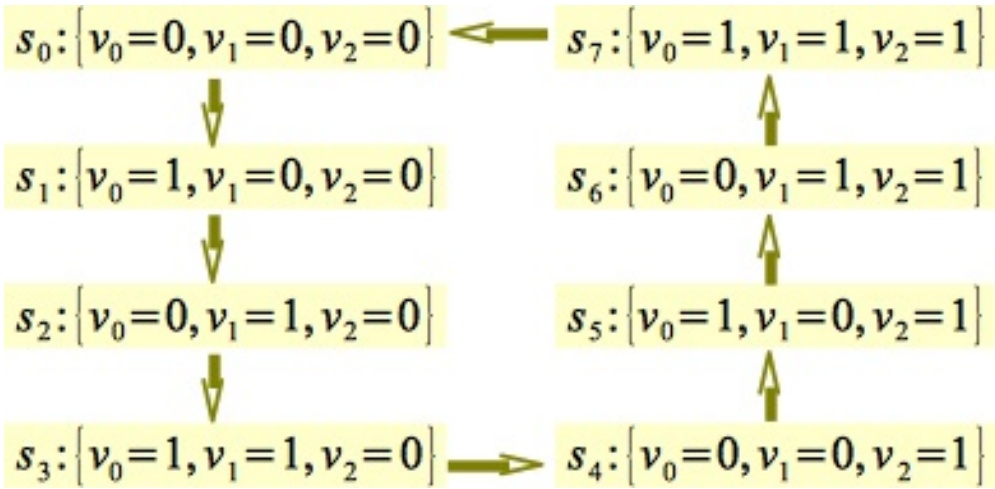
# Programs as Kripke structures

```
1  int x = 0;
2
3  void thread(){
4    int v = x;
5    x = v + 1;
6  }
7
8  void main(){
9    fork(thread);
10   int u = x;
11   x = u + 1;
12   join(thread);
13   assert(x == 2);
14 }
```



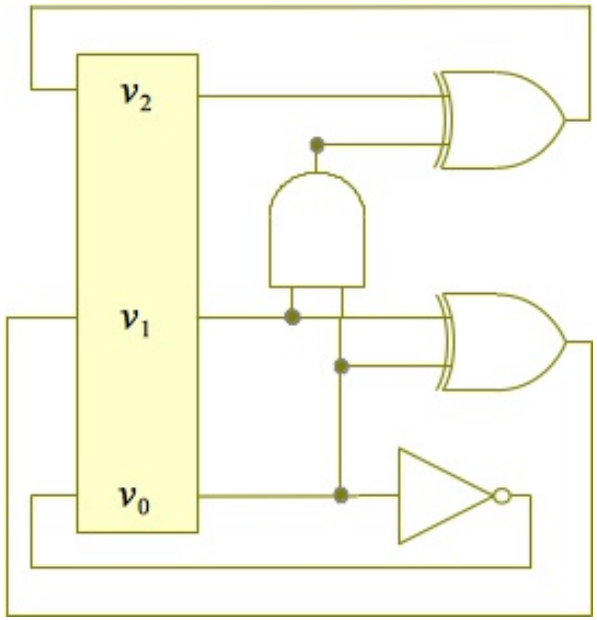
# Synchronous circuits as Kripke structures

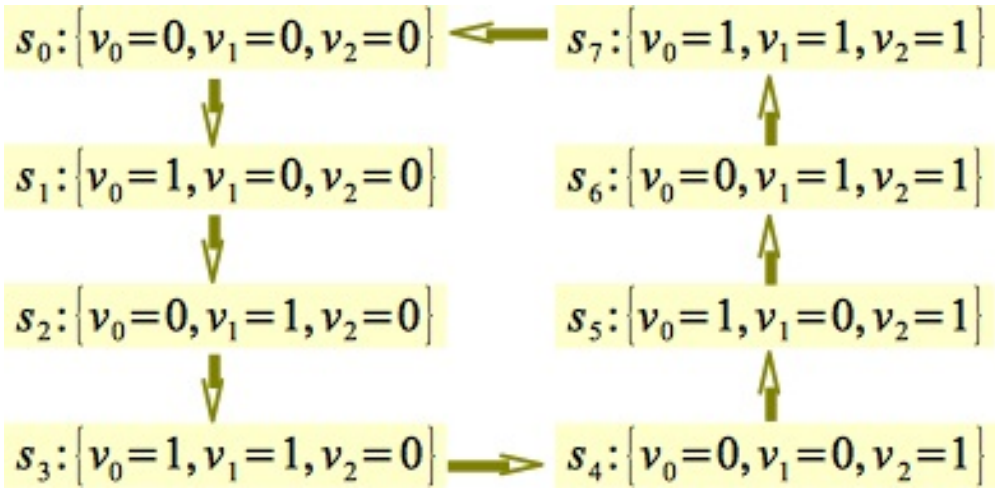



$$v_0' = \neg v_0 \quad (1)$$
$$v_1' = v_0 \oplus v_1 \quad (2)$$
$$v_2' = (v_0 \wedge v_1) \oplus v_2 \quad (3)$$

# Synchronous circuits as Kripke structures




$$v_0' = \neg v_0 \quad (1)$$
$$v_1' = v_0 \oplus v_1 \quad (2)$$
$$v_2' = (v_0 \wedge v_1) \oplus v_2 \quad (3)$$

Asynchronous circuits handled using a disjunctive $R$ instead of a conjunctive one like for synchronous circuits.

# Temporal Logics

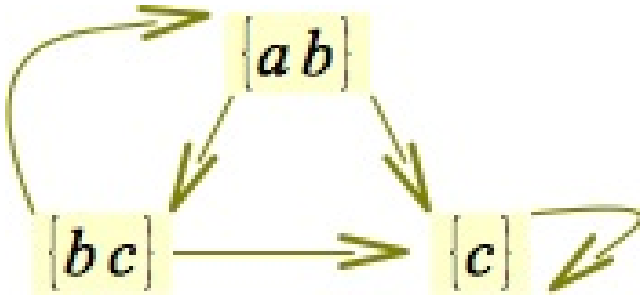
- We are intereseted in describing sequences of transitions of Kripke structures
- Many *Reactive Systems* are designed to continously react to their environement
- An input/output description is not suitable
- Describing sequences makes more sense

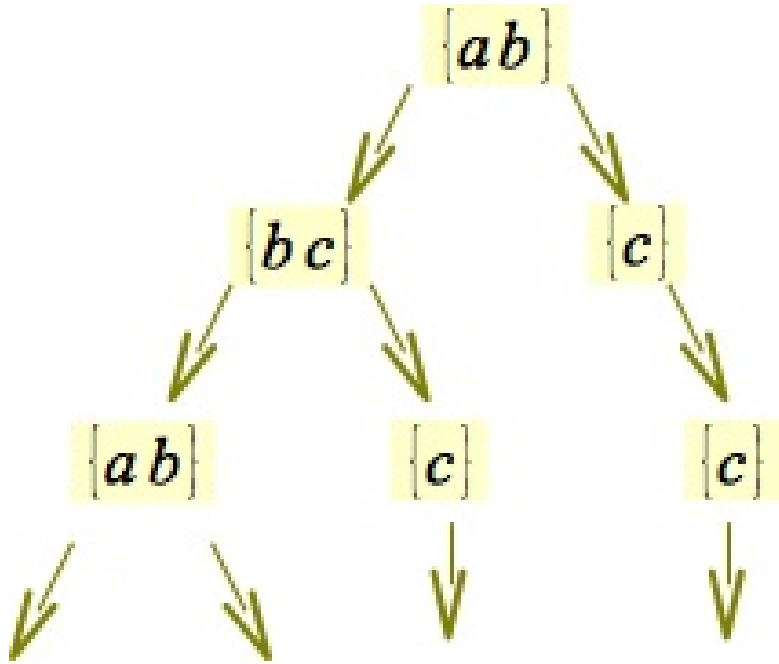
# $\phi$ as a formula in some temporal logics

- Temporal logics are formalisms to describe sequences (hence the notion of time) of transitions
- Temporal operators are used to express that certain properties in $AP$ are:
  - never reached
  - eventually reached
  - more complex combinations of those

# The *CTL*$^*$ Temporal Logic

*Computation trees* are obtained by unwinding the Kripke structure



# The *CTL** Temporal Logic (cont.)

- A *CTL** formua is composed of *path quantifiers* and *temporal operators*
- Path quantifiers (**A**, **E**) describe the branching of the tree. Given a state, **A** (resp. **E**) specify that all (resp. some) path starting at the state have some property
- Temporatl operators (**X**, **F**, **G**, **U**, **R**) describe properties of a given path in the computations tree

# The $CTL^*$ Temporal Logic: syntax

The following are *state fromulas*

- $p$ if $p \in AP$
- $\neg f$, $f \wedge g$ and $f \vee g$ if $f, g$ are state formulas
- $\mathbf{A} f$, $\mathbf{E} f$ if $f$ is a path formula

The following are *path fromulas*

- $f$ if it is also a state formula
- $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\,\mathbf{U}\ G$ and $f\,\mathbf{R}\ g$ if $f, g$ are path formulas

$CTL^*$ is the set of state formulas generated by the above rules.

# The $CTL^*$ Temporal Logic: notation

- A path $\pi = s_0 s_1 \ldots$ in a computation tree (obtained from a Kripke structure) is any infinite sequence of states with $R(s_i, s_{i+1})$ for each $i \in \mathbb{N}$
- Write $\pi^i$ to mean the path starting from $s_i$ in $\pi = s_0 s_1 \ldots$
- Write $M, s \models f$ to mean that state formula $f$ holds at state $s$ in the Kripke structure $M$
- Write $M, \pi \models f$ to mean that path formula $f$ holds along path $\pi$ in the Kripke structure $M$

# The $CTL^*$ Temporal Logic: semantics

$f_1$ and $f_2$ are state formulas, $g_1$ and $g_2$ are path formulas.

$$
\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s) \\
M, s \models \neg f_1 & \Leftrightarrow & M, s \not\models f_1 \\
M, s \models f_1 \vee f_2 & \Leftrightarrow & M, s \models f_1 \text{ or } M, s \models f_2 \\
M, s \models f_1 \wedge f_2 & \Leftrightarrow & M, s \models f_1 \text{ and } M, s \models f_2 \\
M, s \models \mathbf{E}\ g_1 & \Leftrightarrow & \text{there is a path } \pi \text{ from } s \text{ s.t. } M, \pi \models g_1 \\
M, s \models \mathbf{A}\ g_1 & \Leftrightarrow & \text{for every path } \pi \text{ starting from } s,\ M, \pi \models g_1
\end{array}
$$

# The $CTL^*$ Temporal Logic: semantics (cont.)

$f_1$ and $f_2$ are state formulas, $g_1$ and $g_2$ are path formulas.

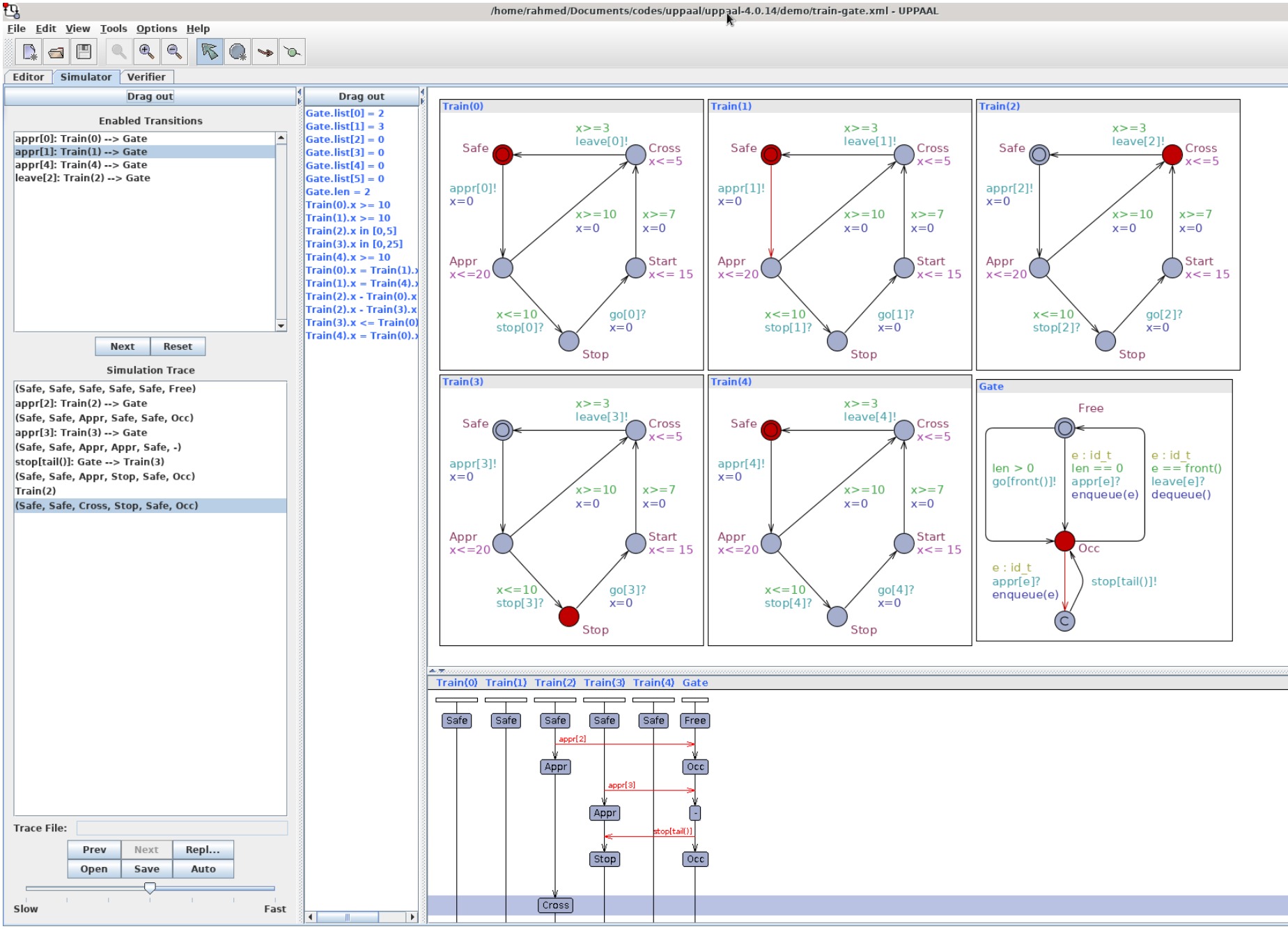
$$
\begin{array}{lcl}
M, \pi \models f_1 & \Leftrightarrow & \text{if } \pi = s_0 s_1 \ldots \text{ then } M, s_0 \models f_1 \\
M, \pi \models \neg g_1 & \Leftrightarrow & M, \pi \not\models g_1 \\
M, \pi \models g_1 \vee g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ or } M, \pi \models g_2 \\
M, \pi \models g_1 \wedge g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ and } M, \pi \models g_2 \\
M, \pi \models \mathbf{X}\ g_1 & \Leftrightarrow & M, \pi^1 \models g_1 \\
M, \pi \models \mathbf{F}\ g_1 & \Leftrightarrow & \text{there exists a } k \geq 0 \text{ s.t. } M, \pi^k \models g_1 \\
M, \pi \models \mathbf{G}\ g_1 & \Leftrightarrow & \text{for all } k \geq 0 \text{ s.t. } M, \pi^k \models g_1 \\
M, \pi \models g_1 \mathbf{U}\ g_2 & \Leftrightarrow & \text{there exists a } k \geq 0 \text{ s.t. } M, \pi^k \models g_2 \\
 & & \text{and for all } 0 \leq j < k, M, \pi^j \models g_1 \\
M, \pi \models g_1 \mathbf{R}\ g_2 & \Leftrightarrow & \text{for all } j \geq 0 \text{ if for every } i < j, M, \pi^i \not\models g_1 \\
 & & \text{then } M, \pi^j \models g_2
\end{array}
$$

# The $CTL^*$ Temporal Logic (cont.)

**Assignment**: Express each of the following using $f, g, \neg, \mathbf{U}, \mathbf{E}$:

- $(\mathbf{F}\ f)\ = ?$
- $(\mathbf{G}\ f)\ = ?$
- $(\mathbf{A}\ f)\ = ?$
- $(f \mathbf{R}\ g)\ = ?$

# The UPPAAL model checker



# Outline

# Further readings



R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.


R. Alur. Timed automata. *International Conference on Computer Aided Verification*, pages 8–22. Springer, 1999.


E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking (chap 2-6)*. MIT press, 2018.

# Further readings



A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.


E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.


D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 163–173, New York, NY, USA, 1980. ACM.


J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982.


E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.