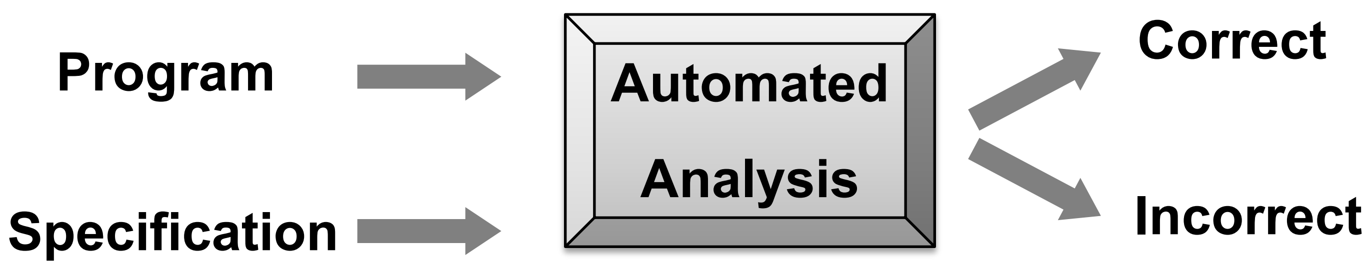
Automated Program Verification with Software Model Checking

Automated Program Verification (APV)
Fall 2018

Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

Static Program Analysis



Reasoning statically about behavior of a program without executing it

- compile-time analysis
- exhaustive, considers all possible executions under all possible environments and inputs

The *algorithmic* discovery of *properties* of program by *inspection* of the *source text*

Manna and Pnueli, “Algorithmic Verification”

Also known as static analysis, program verification, formal methods, etc.

Turing, 1936: “undecidable”

Undecidability

The halting problem

- does a program P terminates on input I
- proved undecidable by Alan Turing in 1936
- https://en.wikipedia.org/wiki/Halting_problem

Rice's Theorem

- for any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property
- in practice, this means that there is no machine that can always decide whether the language of a given Turing machine has a particular nontrivial property
- https://en.wikipedia.org/wiki/Rice%27s_theorem

Living with Undecidability

"Algorithms" that occasionally diverge

Limit programs that can be analyzed
- finite-state, loop-free

Partial (unsound) verification
- analyze only some executions up-to a fixed number of steps

Incomplete verification / Abstraction
- analyze a superset of program executions

Programmer Assistance
- annotations, pre-, post-conditions, inductive invariants

Automated Software Analysis

Model Checking


[Clarke and Emerson, 1981]

[Queille and Sifakis, 1982]

Abstract Interpretation


[Cousot and Cousot, 1977]

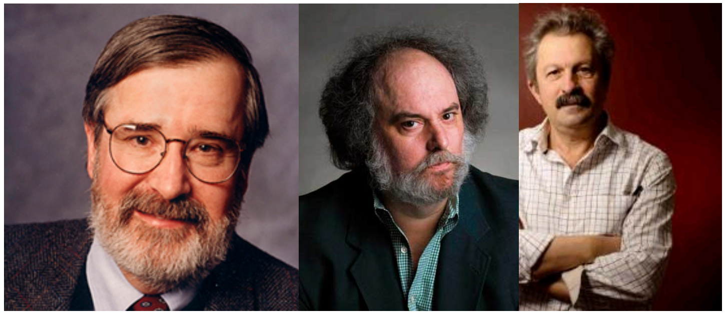
Symbolic Execution

[King, 1976]

(Temporal Logic) Model Checking

Automatic verification technique for finite state concurrent systems.

- Developed independently by Clarke and Emerson and by Queille and Sifakis in early 1980's.
- ACM Turing Award 2007



Specifications are written in propositional temporal logic. (Pnueli 77)

- Computation Tree Logic (CTL), Linear Temporal Logic (LTL), …

Verification procedure is an intelligent exhaustive search of the state space of the design

- Statespace explosion

Model Checking since 1981

1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle

10^5

1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan

10^{100}

1992 SMV: Symbolic Model Verifier
McMillan

1990s: Formal Hardware Verification in Industry: Intel, IBM, Motorola, etc.

1998 Bounded Model Checking using SAT
Biere, Clarke, Zhu

10^{1000}

2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith

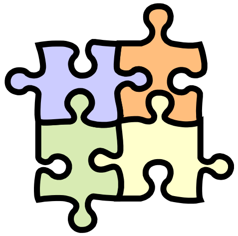
Model Checking since 1981

1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle

1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan

1992 SMV: Symbolic Model Verifier
McMillan



1998 Bounded Model Checking using SAT
Biere, Clarke, Zhu
→ **CBMC**

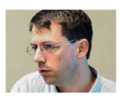
2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith
→ **SLAM, MAGIC, BLAST, …**

2000 started PhD in MC at UofT

multi-valued model checking

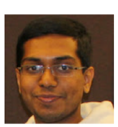
2006 SMC Yasm: safety, liveness, multi-valued abstraction for MC

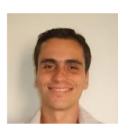
2010 Boxes abstract domain (SAS'10)



2012 UFO: MC + AI: SAS'12

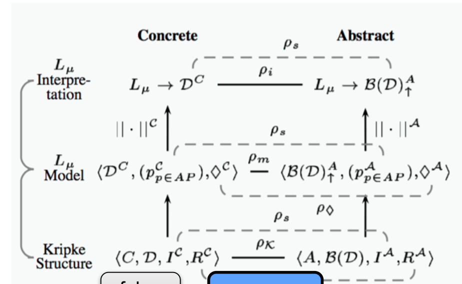
2015 SeaHorn: MC (Spacer) **and** AI (Crab)

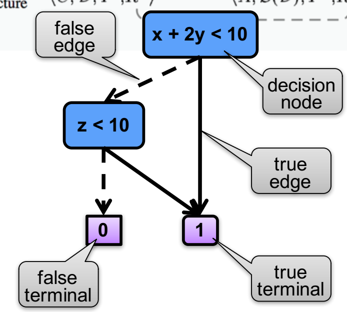


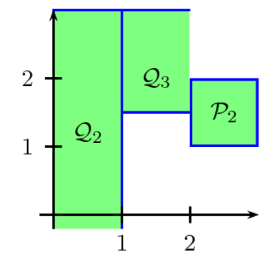


SLAM BLAST VMCAI CBMC

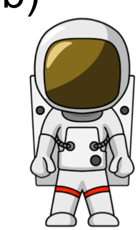
VMCAI'06



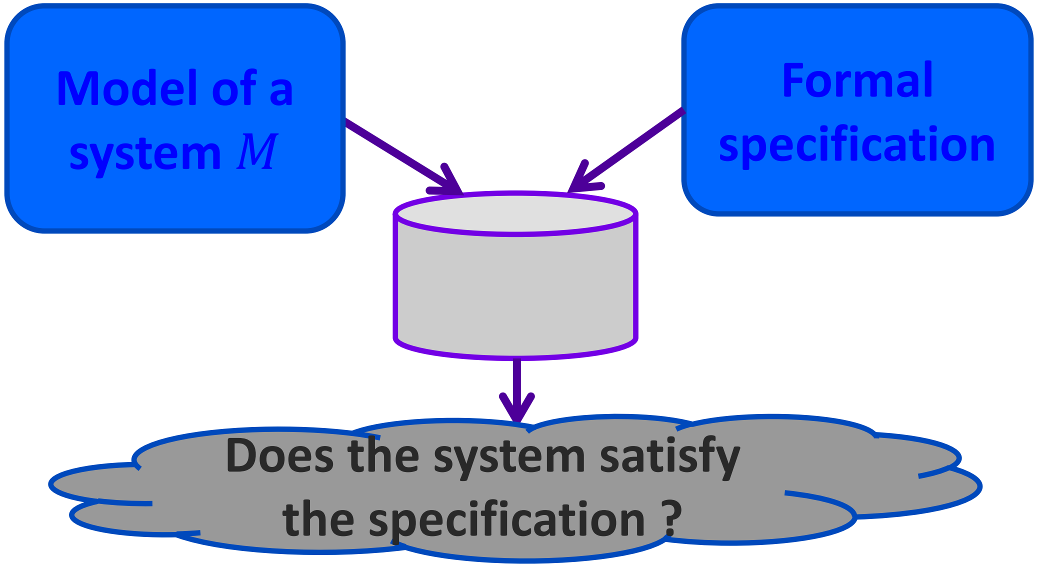




UFO CPAchecker SMACK SV-COMP



Classical Model Checking* [EC81,QS82]



Not decidable!

To enable automation, **Model Checking** restricts the problem:

Model: **Finite-state** reactive systems

Specification: Propositional **temporal logics**

*Clarke, Emerson, and Sifakis won the 2007 Turing award for their contribution to MC

UNIVERSITY OF WATERLOO

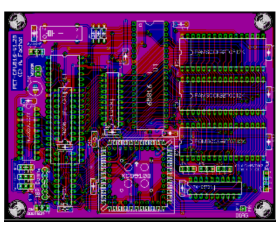
Finite State Reactive Systems - Examples

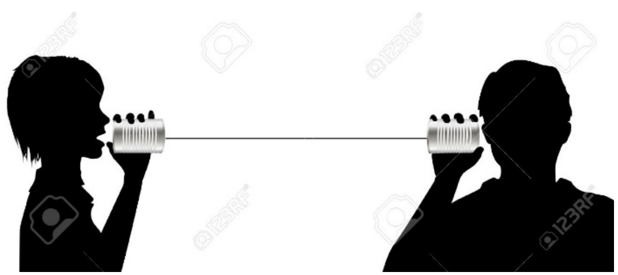
Hardware designs

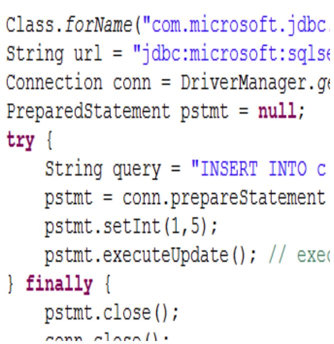
Controllers (elevator, traffic-light)

Communication protocols (when ignoring the message content)

High level (abstracted) description of infinite state systems



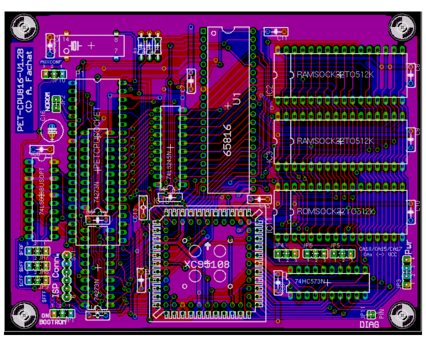


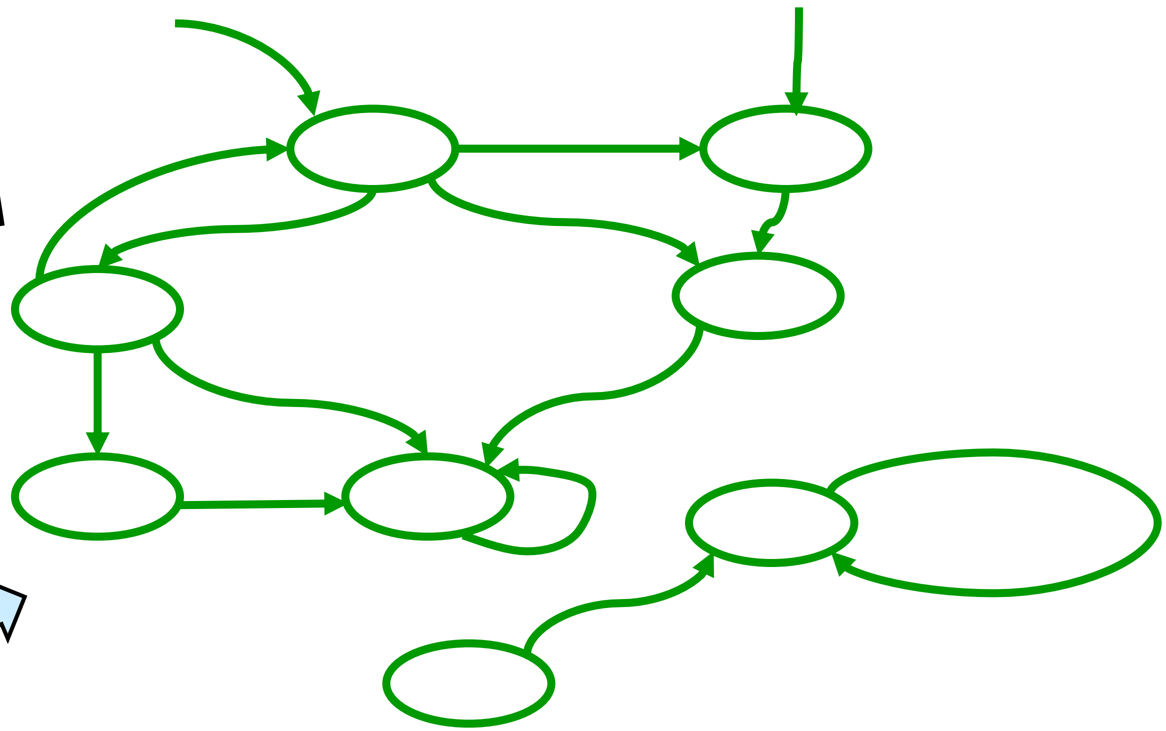


Model of a system

Kripke structure / transition system

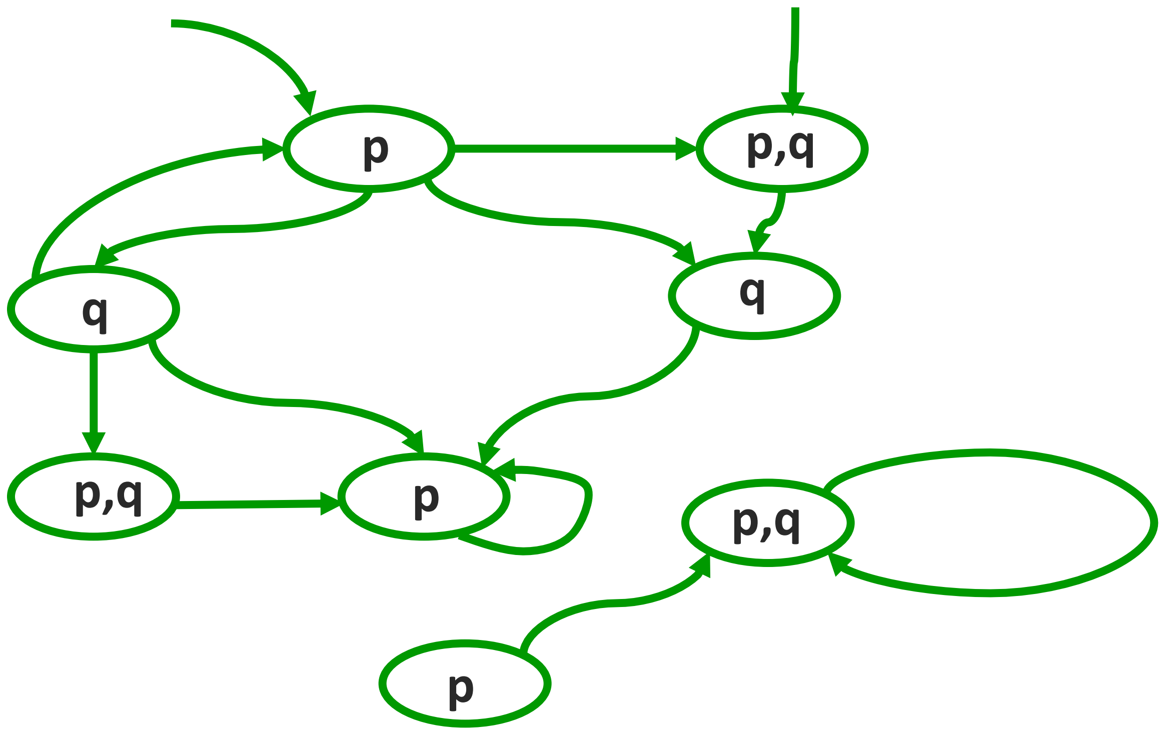
```
Class.forName("com.microsoft.jdbc.
String url = "jdbc:microsoft:sqlse
Connection conn = DriverManager.ge
PreparedStatement pstmt = null;
try {
    String query = "INSERT INTO c
    pstmt = conn.prepareStatement
    pstmt.setInt(1,5);
    pstmt.executeUpdate(); // exec
} finally {
    pstmt.close();
```





Model of a system (cont.)

Kripke structure / transition system



States labeled by **atomic propositions (AP)**

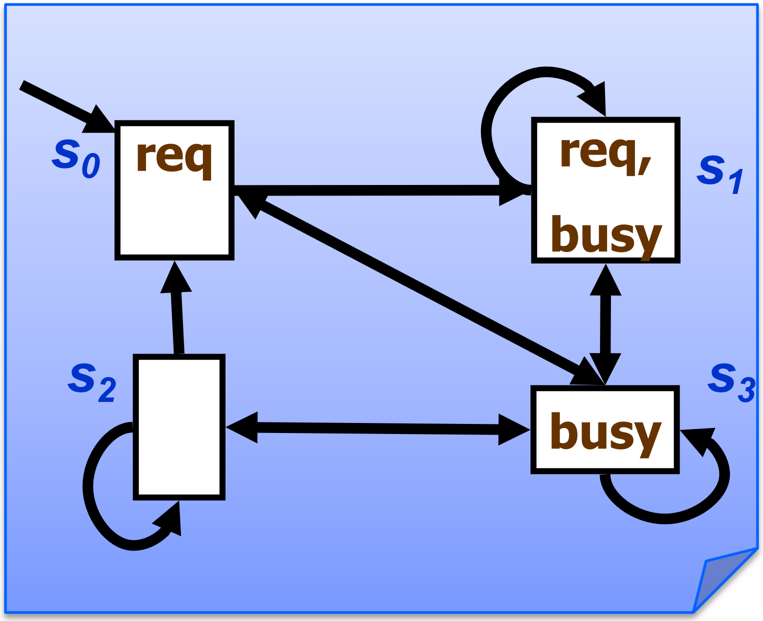
- "x=0",
- "Printer is busy",
- "process in critical section",
- …

Reactive systems:

Set of **states** is **finite**,

But **computations** are **infinite**

Models: Kripke Structures



Conventional state machines

- $K = (V, S, s_0, I, R)$
- V is a (finite) set of atomic propositions
- S is a (finite) set of states
- $s_0 \in S$ is a start state
- $I: S \rightarrow 2^V$ is a labelling function that maps each state to the set of propositional variables that hold in it
  - That is, $I(S)$ is a set of interpretations specifying which propositions are true in each state
- $R \subseteq S \times S$ is a transition relation

From Programs to Kripke Structures

Program

```
1:  int x = 2;
    int y = 2;
2:  while (y <= 2)
3:    y = y - 1;
4:  if (x == 2)
5:      x =1;
6:
```

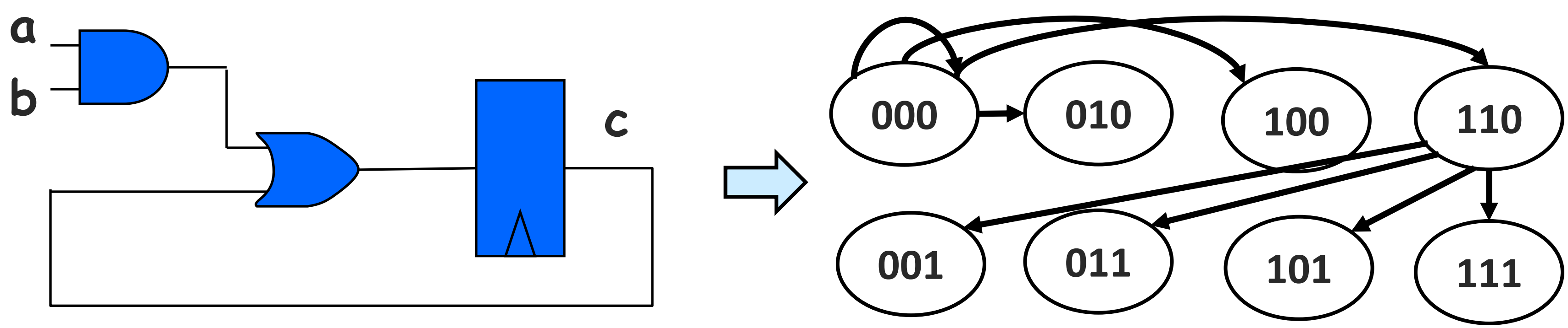
State

pc	x	y	…
3	1	3	…

Transition

pc	x	y	…
2	1	2	…

From Circuits to Kripke Structures



States = valuations to variables a,b,c

→ 8 states: 000,001,…

Transitions:

a,b: inputs, change arbitrarily

c: state variable, updated according to circuit

$c' \leftrightarrow (a \wedge b) \vee c$

Modal Logic

Extends *propositional logic* with modalities to qualify propositions

- *"it is raining"* – *rain*
- "it will rain tomorrow" – $\Box$*rain*
  - it is raining in all possible futures
- "it might rain tomorrow" – $\Diamond$*rain*
  - it is raining in some possible futures

Modal logic formulas are interpreted over a collection of *possible worlds* connected by an *accessibility relation*

Temporal logic is a modal logic that adds temporal modalities: next, always, eventually, and until

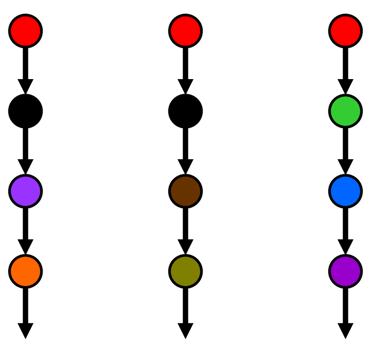
Temporal Logic

[A. Pnueli, FOCS 1977]

- **Temporal Logics**
  - **Express properties of event orderings in time**

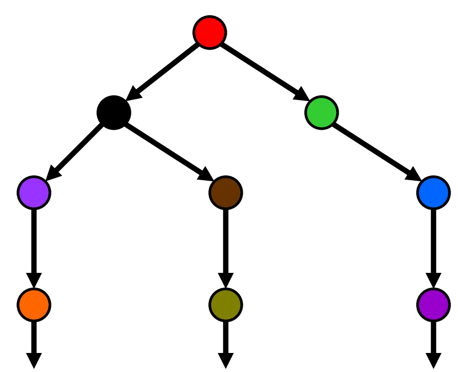
Linear Time

- Every moment has a unique successor
- Infinite sequences (words)
- Linear Time Temporal Logic (LTL)



Branching Time

- Every moment has several successors
- Infinite tree
- Computation Tree Logic (CTL)



UNIVERSITY OF WATERLOO

Propositional temporal logic

AP – a set of atomic propositions

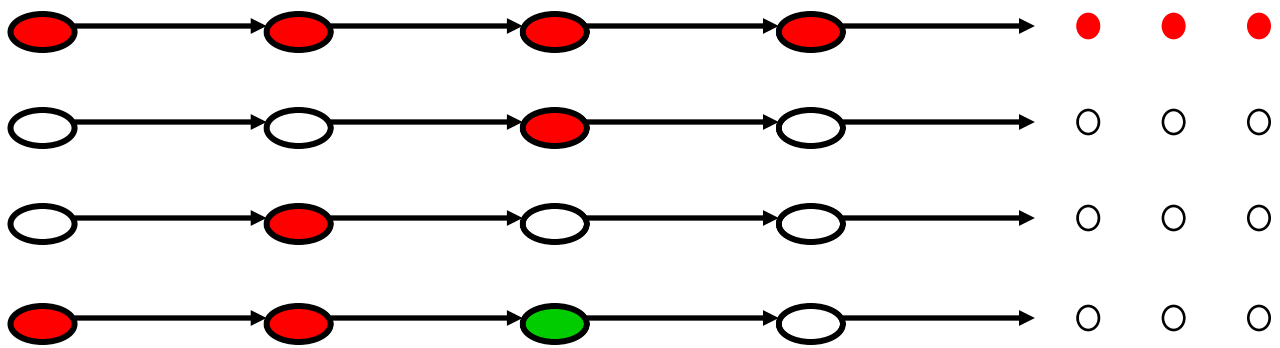
Temporal operators:

Gp

Fp

Xp

pUq



Path quantifiers: A for **all** path

E there **exists** a path

LTL/CTL/CTL*

LTL – of the form **A**ψ

ψ **-** path formula, contains **no** path **quantifiers** but any nesting of temporal operators

interpreted over infinite computation paths

CTL – path quantifiers and temporal operators appear in pairs: **AG, AU, AF, AX, EG, EU, EF, EX**

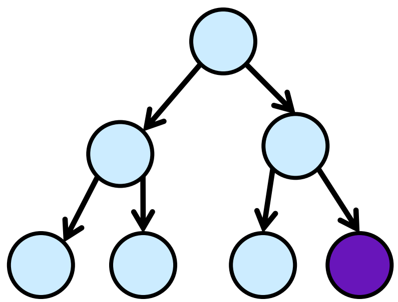
interpreted over infinite computation trees

CTL* - Allows any combination of temporal operators and path quantifiers. Includes both LTL and CTL

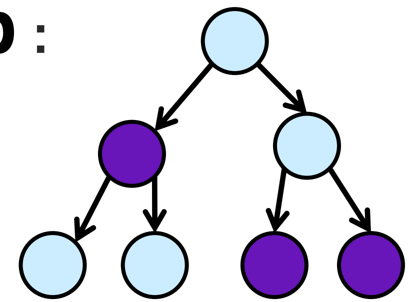
Illustration of CTL Semantics

EFp :

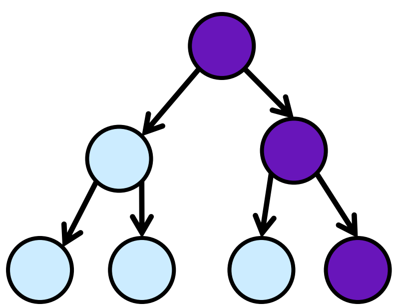
“exists reachable state s.t.”



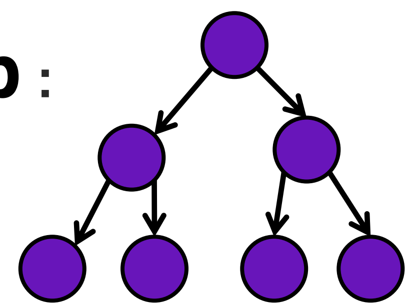
AFp :



EGp :



AGp :



“all reachable states….”

Properties in Temporal Logic - Examples

CTL formulas:

mutual exclusion: **AG** $\neg(cs_1 \wedge cs_2)$

non starvation: **AG** (request ⇒ **AF** grant)

"sanity" check: **EF** request

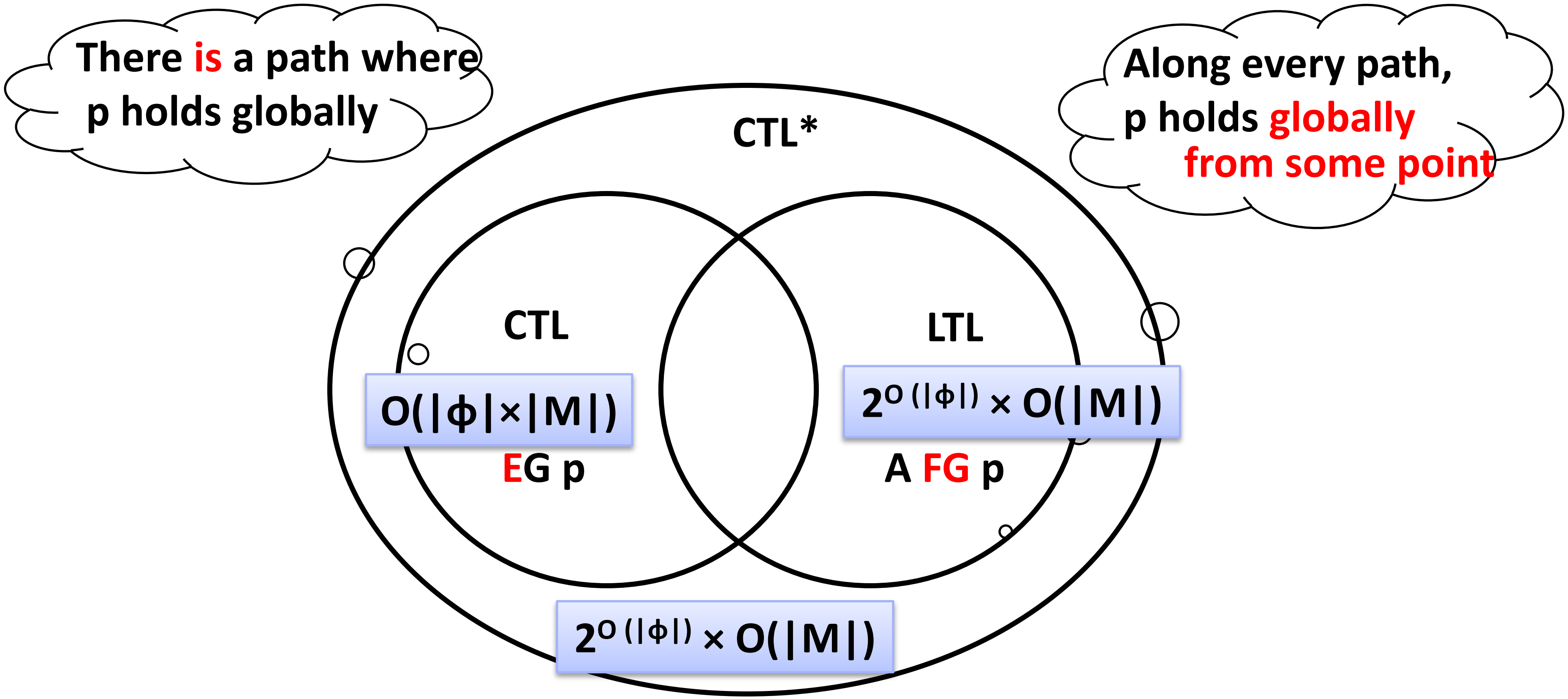
Communication protocols: **A** (¬get-msg) **U** send-msg

LTL formulas:

fairness: **A**(**GF** enabled ⇒ **GF** executed)

A(x=a ∧ y=b ⇒ **XXXX** z=a+b)

LTL/CTL/CTL*



ACTL / ACTL*: The universal fragments of CTL/CTL* with only universal path quantifiers

UNIVERSITY OF WATERLOO

Some Statements To Express

An elevator can remain idle on the third floor with its doors closed

- EF (state=idle $\wedge$ floor=3 $\wedge$ doors=closed)

When a request occurs, it will eventually be acknowledged

- AG (request $\Rightarrow$ AF acknowledge)

A process is enabled infinitely often on every computation path

- AG AF enabled

A process will eventually be permanently deadlocked

- AF AG deadlock

Action *s* precedes *p* after *q*

- A[$\neg q$ U ($q \wedge$ A[$\neg p$ U s])]
- Note: hard to do correctly. Use property patterns

Expressing Properties in LTL

Good for safety (G $\neg$) and liveness (F) properties

Express:

- When a request occurs, it will eventually be acknowledged
  - G (request $\Rightarrow$ F acknowledge)
- Each path contains infinitely many *q*'s
  - G F *q*
- At most a finite number of states in each path satisfy $\neg q$ (or property *q* eventually stabilizes)
  - F G *q*
- Action *s* precedes *p* after *q*
  - $[\neg q \; U \; (q \wedge [\neg p \; U \; s])]$
  - Note: hard to do correctly.

Safety and Liveness

Safety AG ¬bad

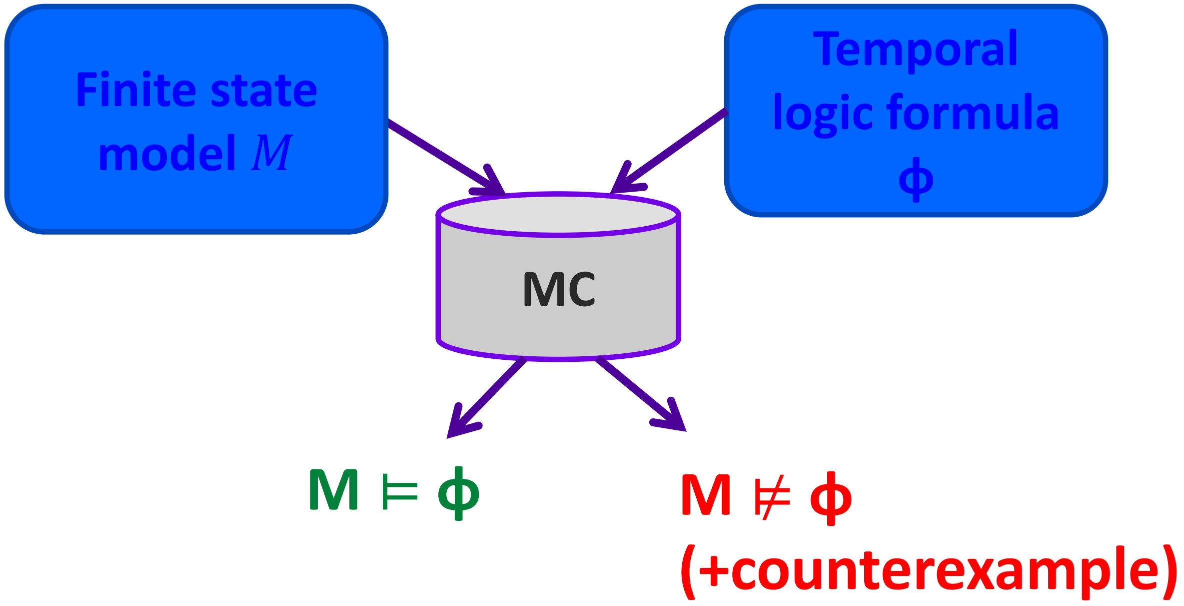
- e.g., mutual exclusion: no two processes are in their critical section at once
- if false then there is a finite cex
- Safety = reachability

Liveness AF good

- e.g., every request is eventually serviced
- if false then there is an infinite cex
- Liveness = termination

* Every LTL formula can be decomposed into a safety property and a liveness property

Model Checking

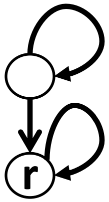


Property types

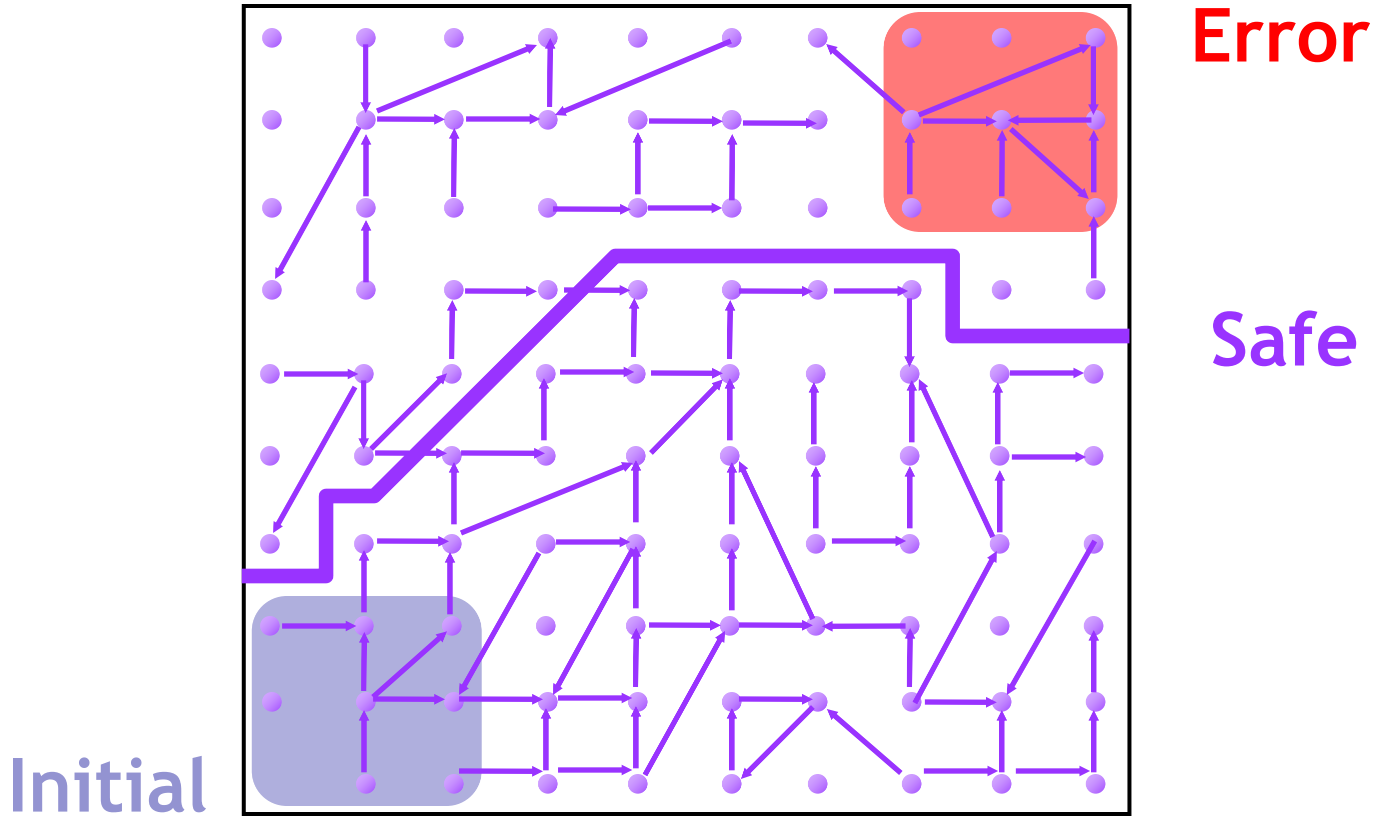
	Universal	Existential
Safety	AG ¬bad • e.g., mutual exclusion: no two processes are in their critical section at once • if false then there is a finite cex • Safety = reachability	EG ¬bad
Liveness	AF good • e.g., every request is eventually serviced • if false then there is an infinite cex • Liveness = termination	EF good

Combinations: AG EF reset

"along every possible execution, in every state there is a possible continuation that will eventually reach a reset state"



The Safety Verification Problem



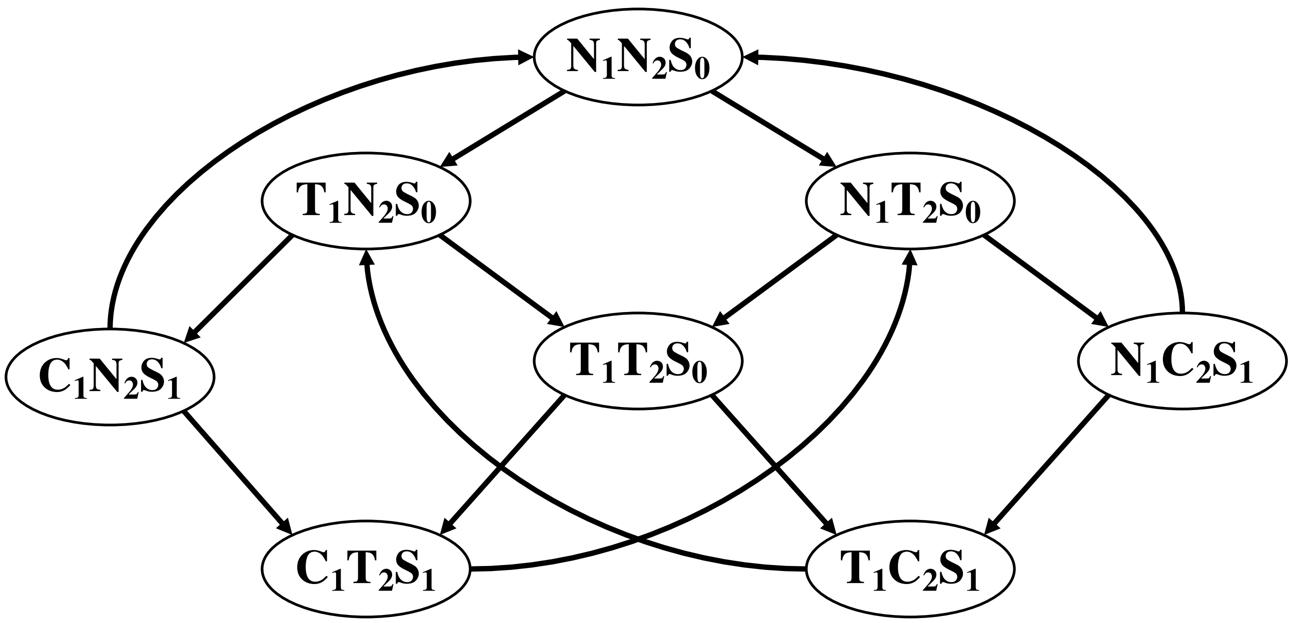
Is there a path from an initial to an error state?

Mutual Exclusion Example
[by Willem Visser]

- Two process mutual exclusion protocol with shared semaphore
- Each process has three states
  - Non-critical (N)
  - Trying (T)
  - Critical (C)
- Semaphore can be available (S_0) or taken (S_1)
- Initially both processes are in the Non-critical state and the semaphore is available --- $N_1 N_2 S_0$

$$
\begin{array}{lcl}
N_1 & \rightarrow & T_1 \\
T_1 \wedge S_0 & \rightarrow & C_1 \wedge S_1 \\
C_1 & \rightarrow & N_1 \wedge S_0
\end{array}
\quad || \quad
\begin{array}{lcl}
N_2 & \rightarrow & T_2 \\
T_2 \wedge S_0 & \rightarrow & C_2 \wedge S_1 \\
C_2 & \rightarrow & N_2 \wedge S_0
\end{array}
$$

Model for Mutual Exclusion

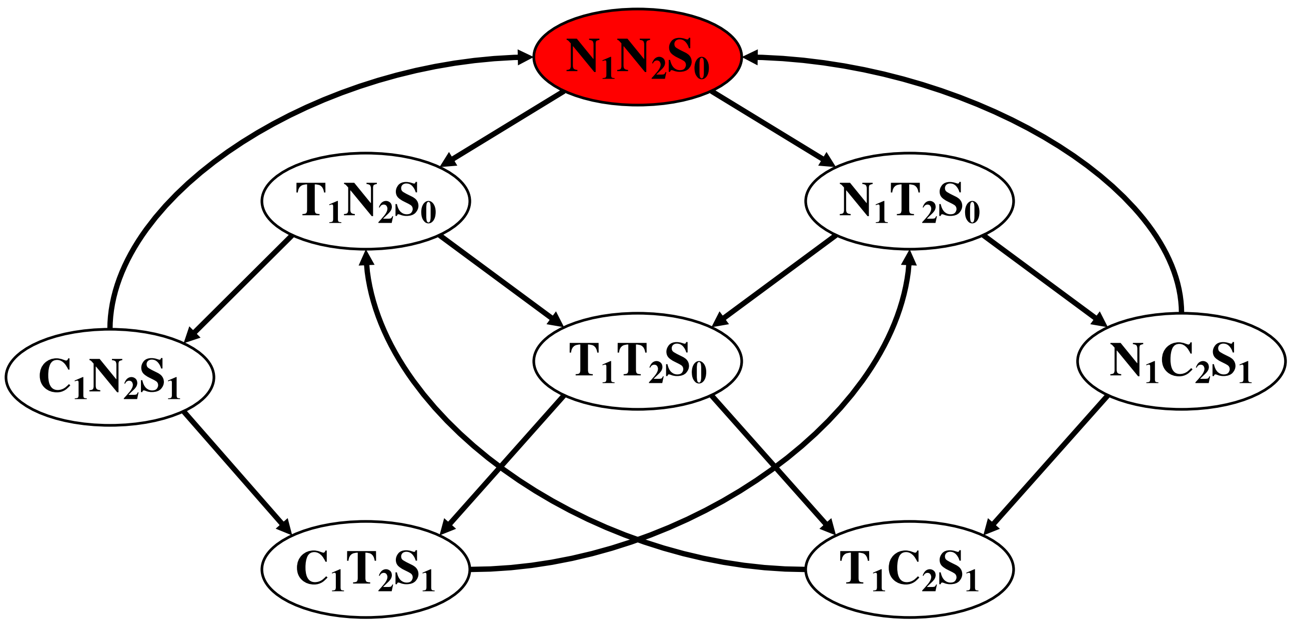


Specification: $M \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

No matter where you are there is always a way to get to the initial state

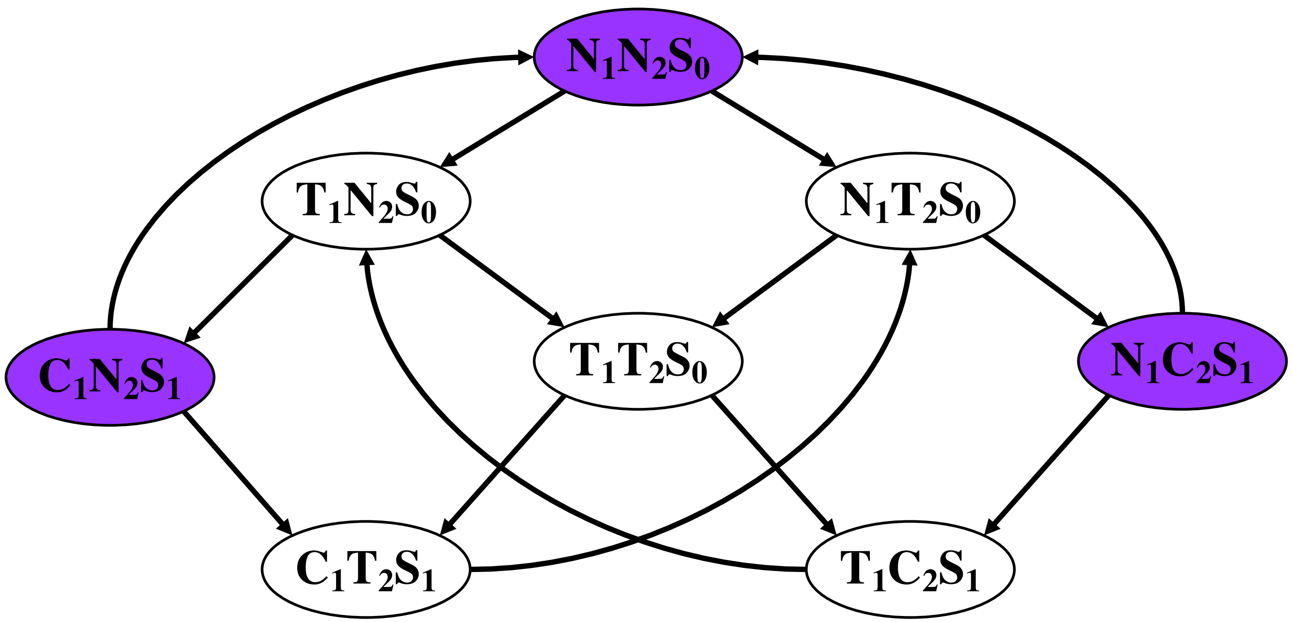
UNIVERSITY OF WATERLOO

Mutual Exclusion Example



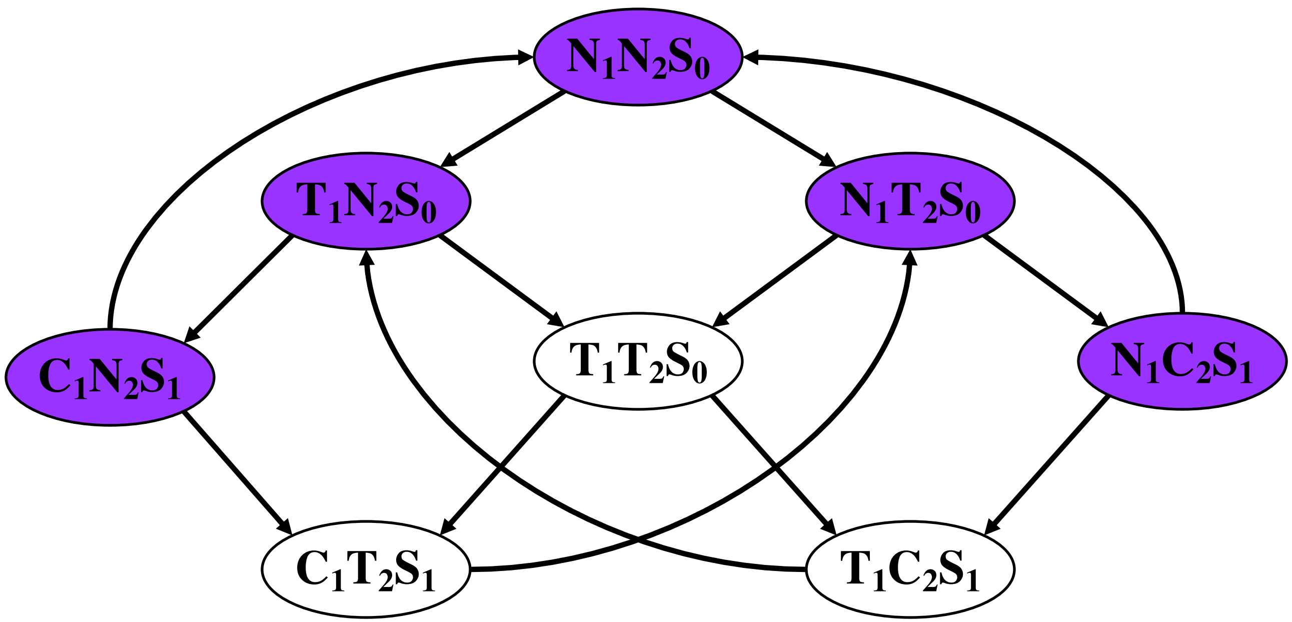
$$M \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



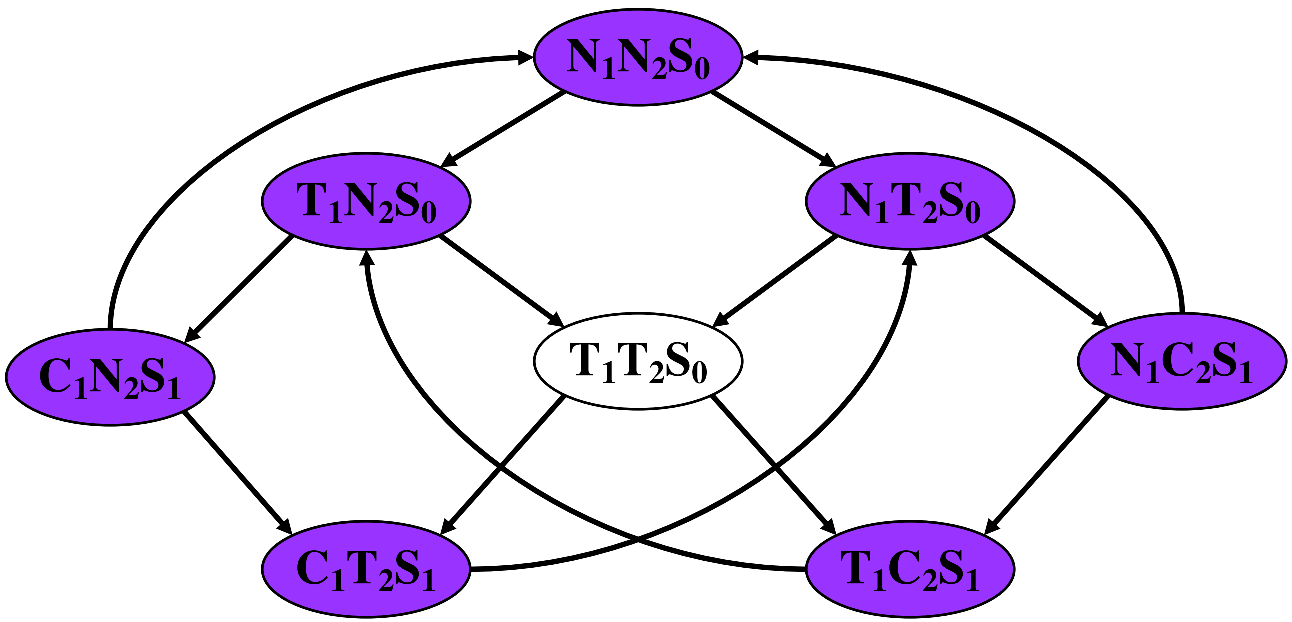
$$M \not\models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



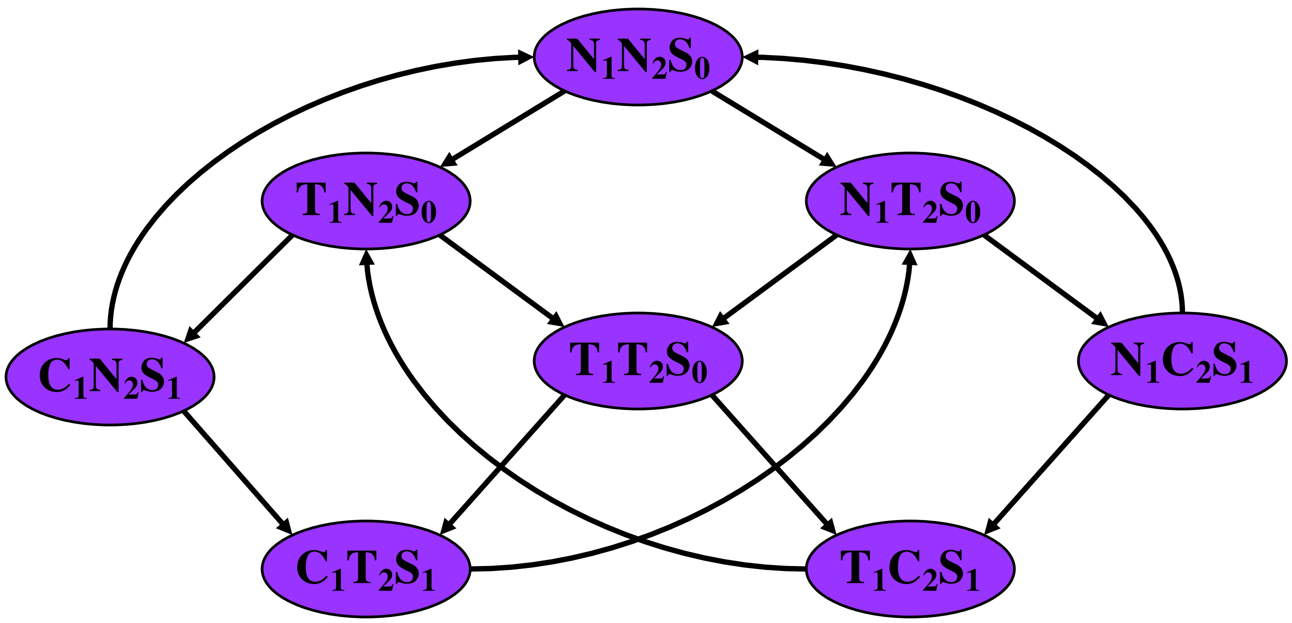
$$M \not\models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



$$M \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

Mutual Exclusion Example



$$M \not\models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$$

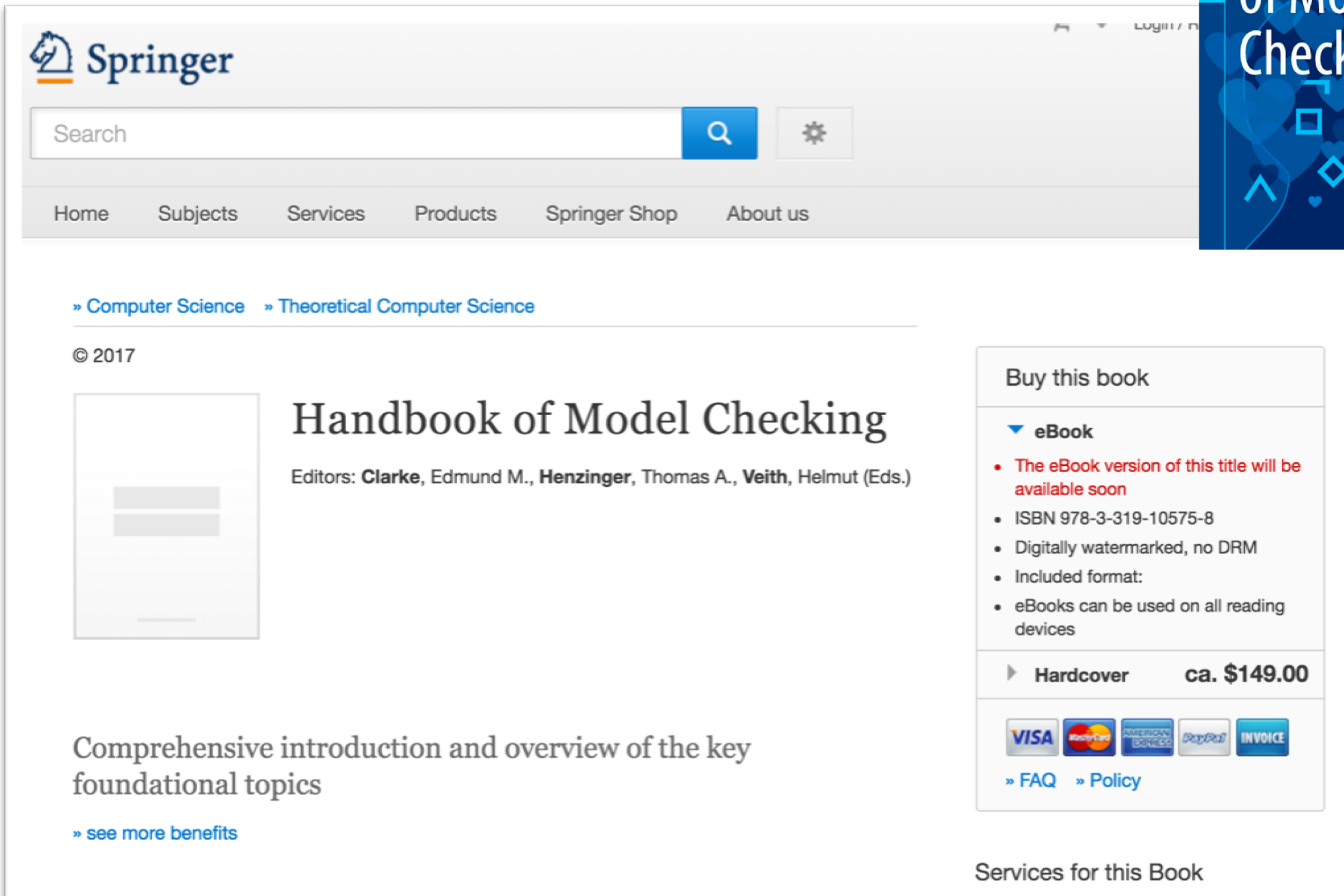
No matter where you are there is always a way to get to the initial state

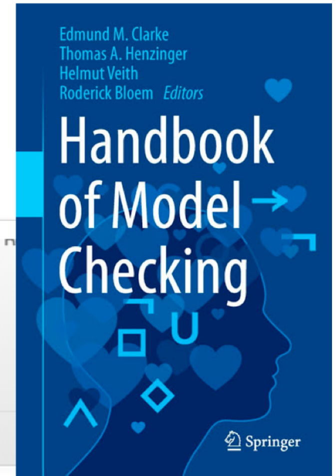
Applications of Model Checking

- Emerging as an industrial standard for verification of hardware designs: Intel, IBM, Cadence, Synopsis, …
  - HWMCC: annual competition of academic tools (http://fmv.jku.at/hwmcc15/)

- Emerging as software verification:
  - Industry: SLAM (Microsoft), F-Soft (NEC), …
  - Academic tools: CBMC, BLAST, UFO, CPAChecker, Smack, SeaHorn, …
  - SV-COMP: annual Software Verification competition (http://sv-comp.sosy-lab.org/2018/)

Handbook of Model Checking (2017)





Handbook of Model Checking (2017)

What Is Model Checking?

Temporal Logic and Fair Discrete Systems.

Modeling.

Binary Decision Diagrams.

Propositional SAT Solving.

Procedures for Satisfiability Modulo Theories.

Automata Theory and Model Checking.

The mu-calculus as a Formalism for Verification.

BDD-Based Symbolic Model Checking.

SAT-Based Model Checking.

Explicit-State Model Checking.

Partial-Order Reduction.

Abstraction and Abstraction-Refinement.

Compositional Reasoning.

Interpolation: Proofs in the Service of Model Checking.

Model Checking and Deduction.

Transfer of Model Checking Theory to Industrial Practice.

Property Specification Languages for Hardware.

Predicate Abstraction for Program Verification

Model Checking Concurrent Software.

Combining Model Checking and Data-Flow Analysis.

Combining Model Checking and Testing.

Symbolic Trajectory Evaluation.

Model Checking Procedural Programs.

Parameterized Systems.

Model Checking Security Protocols.

Games and Synthesis.

Symbolic Model Checking in Non-Bool. Domains.

Verification of Real-Time Systems.

Verification of Hybrid Systems.

Probabilistic Model Checking.

Model Checking and Process Algebra.

State Explosion

How fast do Kripke structures grow?

- Composing linear number of structures yields exponential growth!

How to deal with this problem?

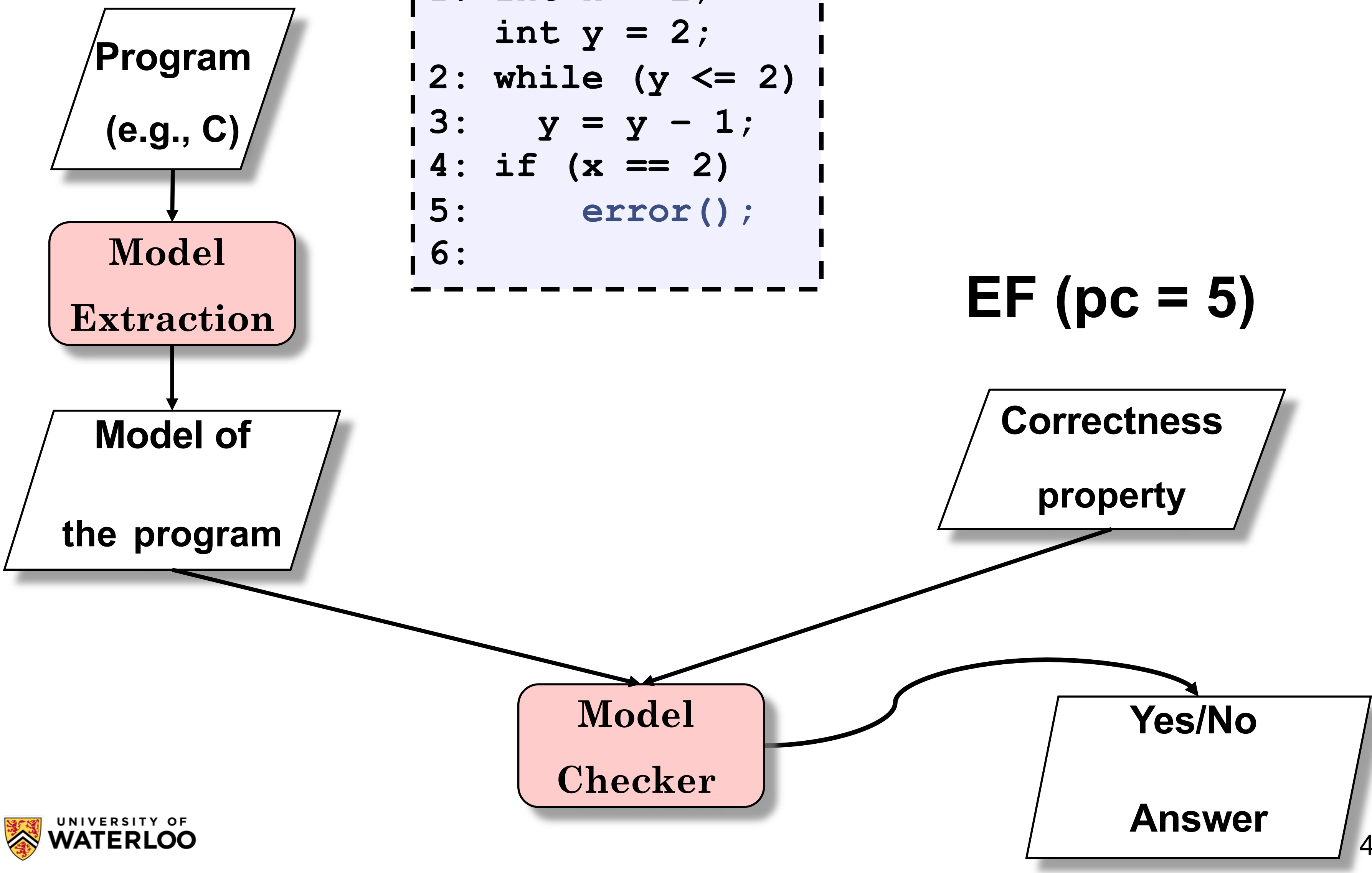
- Symbolic model checking with efficient data structures (BDDs, SAT).
  - Do not need to represent and manipulate the entire model
- Abstraction
  - Abstract away variables in the model which are not relevant to the formula being checked
  - Partial order reduction (for asynchronous systems)
  - Several interleavings of component traces may be equivalent as far as satisfaction of the formula to be checked is concerned
- Composition
  - Break the verification problem down into several simpler verification problems

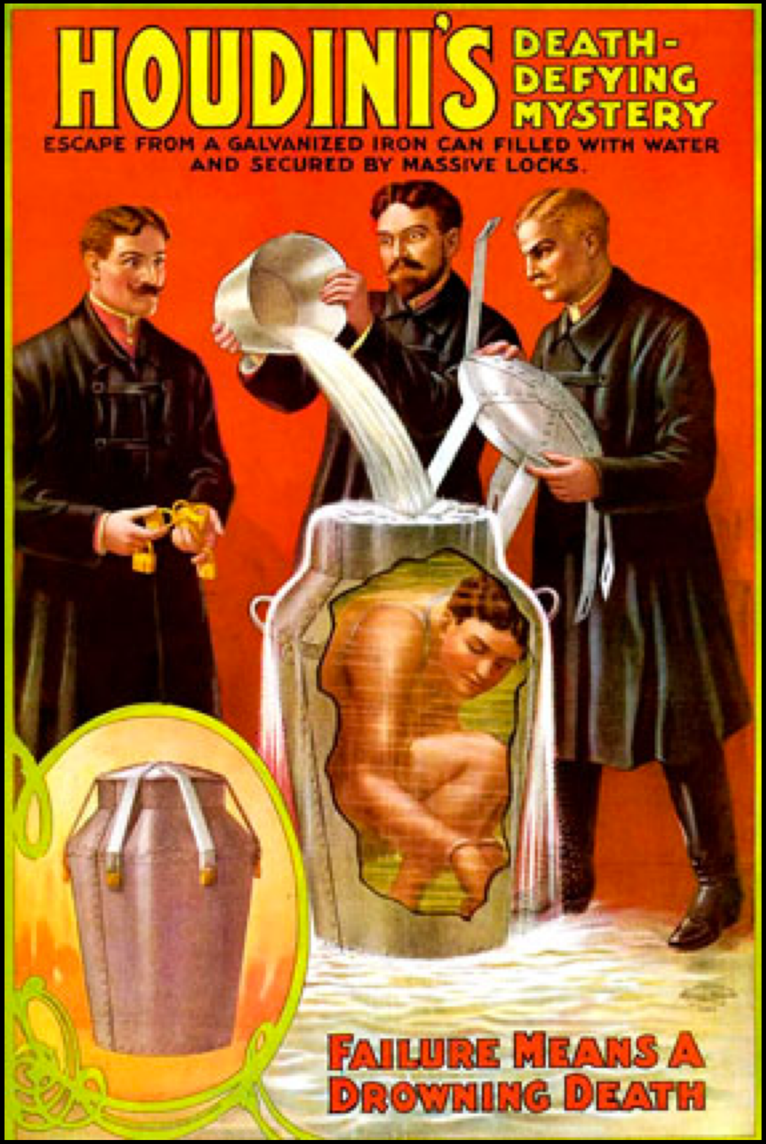
SOFTWARE MODEL CHECKING

Software Model Checking

```
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:   y = y - 1;
4: if (x == 2)
5:     error();
6:
```

EF (pc = 5)



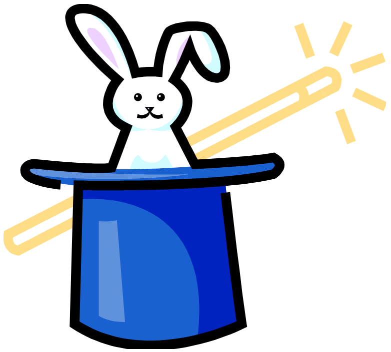

HOUDINI'S DEATH-DEFYING MYSTERY
ESCAPE FROM A GALVANIZED IRON CAN FILLED WITH WATER AND SECURED BY MASSIVE LOCKS.
FAILURE MEANS A DROWNING DEATH

A Magician's Guide to Solving Undecidable Problems

Develop a procedure ***P*** for a decidable problem

Show that ***P*** is a decision procedure for the problem

- e.g., model checking of finite-state systems

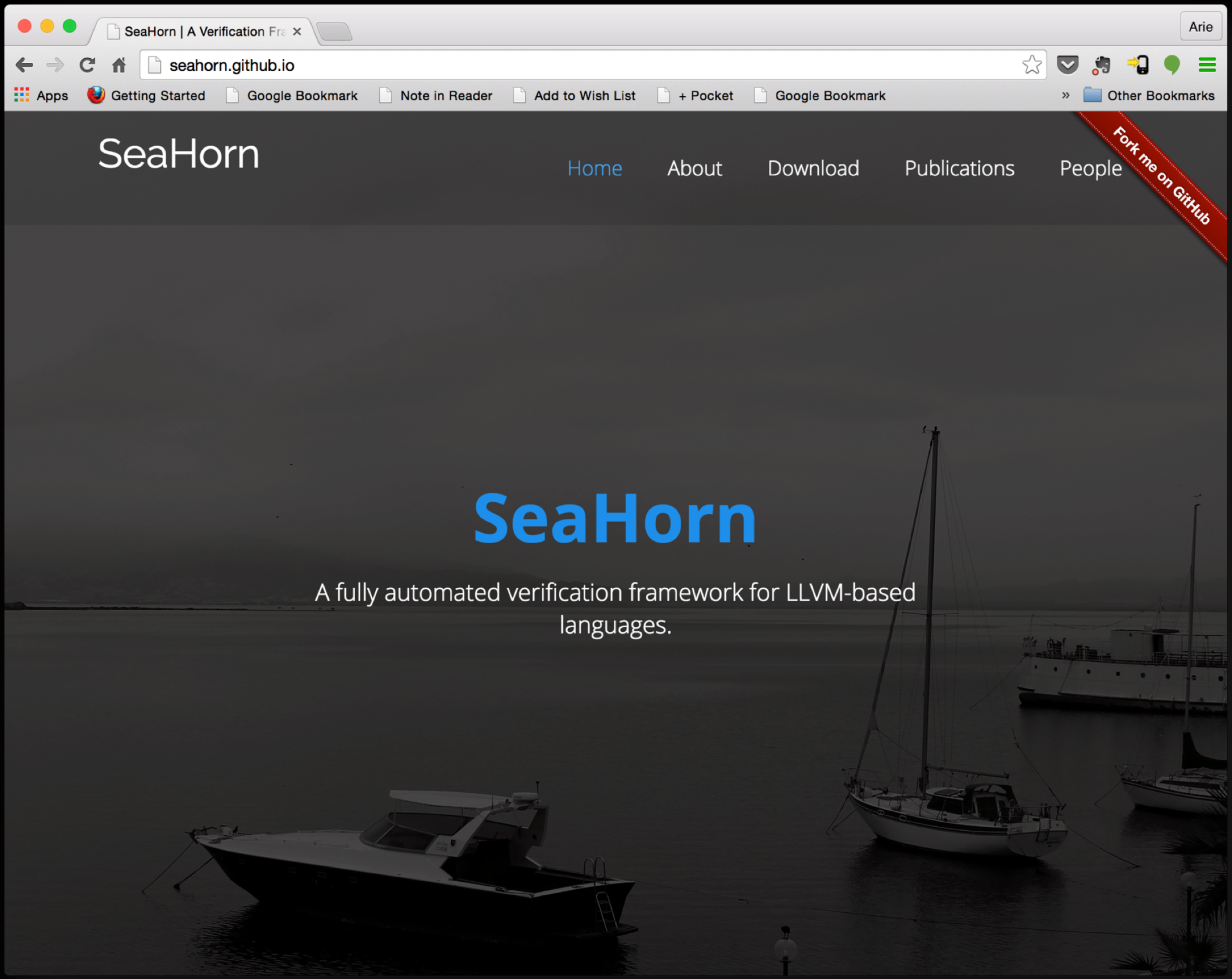


Choose one of

- Always terminate with some answer (over-approximation)
- Always make useful progress (under-approximation)

Extend procedure ***P*** to procedure ***Q*** that "solves" the undecidable problem

- Ensure that ***Q*** is still a decision procedure whenever ***P*** is
- Ensure that ***Q*** either always terminates or makes progress



http://seahorn.github.io

SeaHorn Usage

Example: in test.c, check that x is always greater than or equal to y

test.c

```c
extern int nd();
extern void __VERIFIER_error() __attribute__((noreturn));
void assert (int cond) { if (!cond) __VERIFIER_error (); }
int main(){
  int x,y;
  x=1; y=0;
  while (nd ())
  {
    x=x+y;
    y++;
  }
  assert (x>=y);
 return 0;
}
```

SeaHorn command:

```
)-> sea pf test.c
```

SeaHorn result:

```
            SEAHORN
--------------------------------
PROPERTY (line 12) | TRUE
--------------------------------
TIME(ms)           |  0.06
```

SeaHorn at a glance

Publicly Available (http://seahorn.github.io)
state-of-state-of-the-art Software Model Checker

Industrial-strength front-end based on Clang and LLVM

Abstract Interpretation engine: Crab

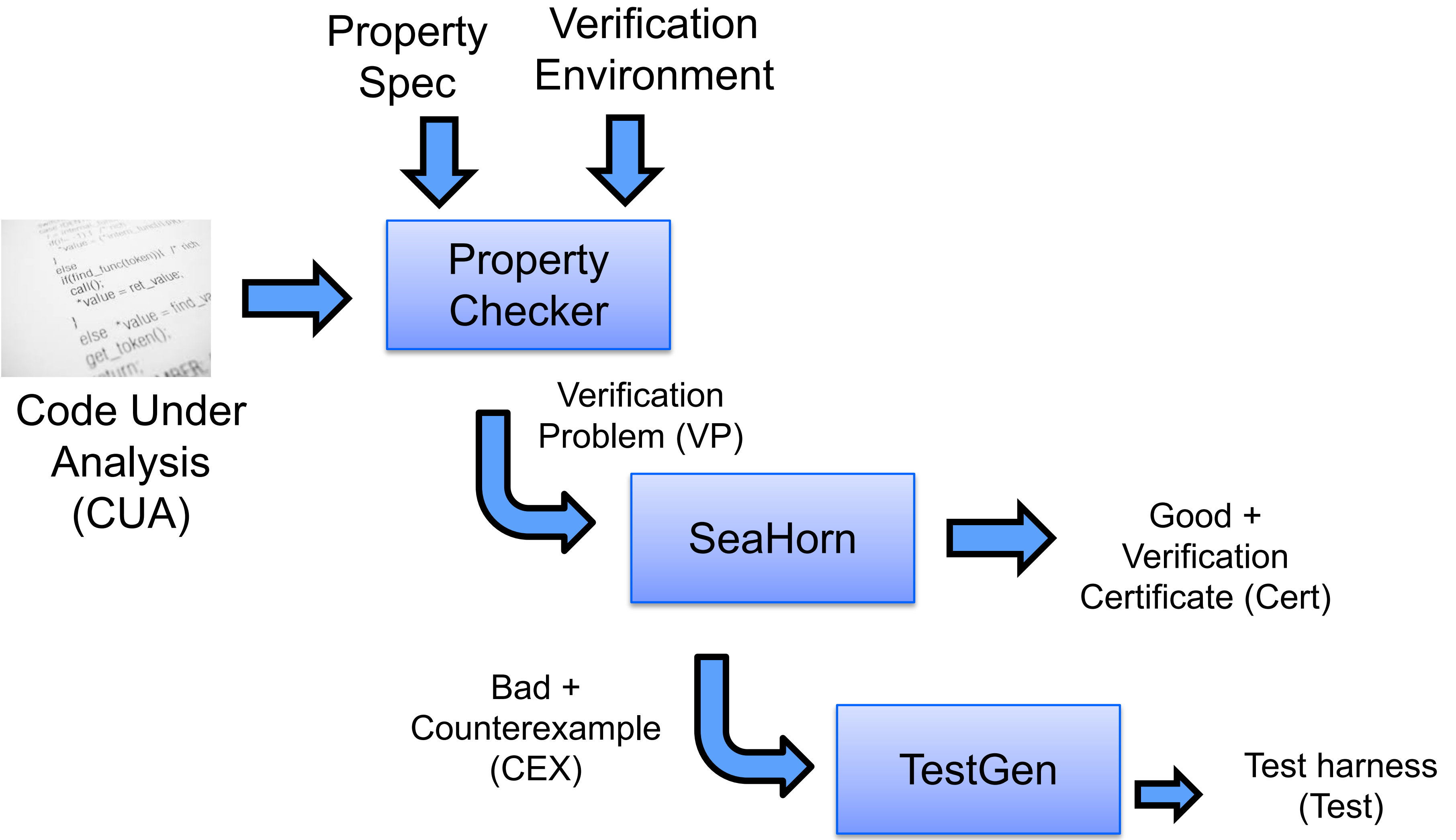
SMT-based verification engine: Spacer

Bit-precise Bounded Model Checker and Symbolic Execution

Executable Counter-Examples

A framework for research and application of logic-based verification

SeaHorn Workflow



SeaHorn workflow components

Code Under Analysis (CUA)

- code being analyzed. Device driver, component, library, etc.

Verification Environment

- stubs for the environment with which CUA interacts
- e.g., libc, memcpy, malloc, OS system calls, user input, socket, file, …

Property Checker

- static instrumentation of a program with a monitor that indicates when an error has happened
- similar to dynamic sanitizers, but can use verifier-specific API to perform symbolic actions
- property spec is specific to a property checker

Verification Problem

- a prepared instance of program with embedded assertions, potentially simplified by abstracting away irrelevant parts of execution

Test Gen

- generates a test harness that includes all stubs and stimuli to guide CUA to a property failure discovered by the verifier

Developing a Static Property Checker

A static property checker is similar to a dynamic checker

- e.g., clang sanitizer (address, thread, memory, etc.)

A significant development effort for each new property

- new specialized static analyses to rule out trivial cases
- different instrumentations have affect on performance

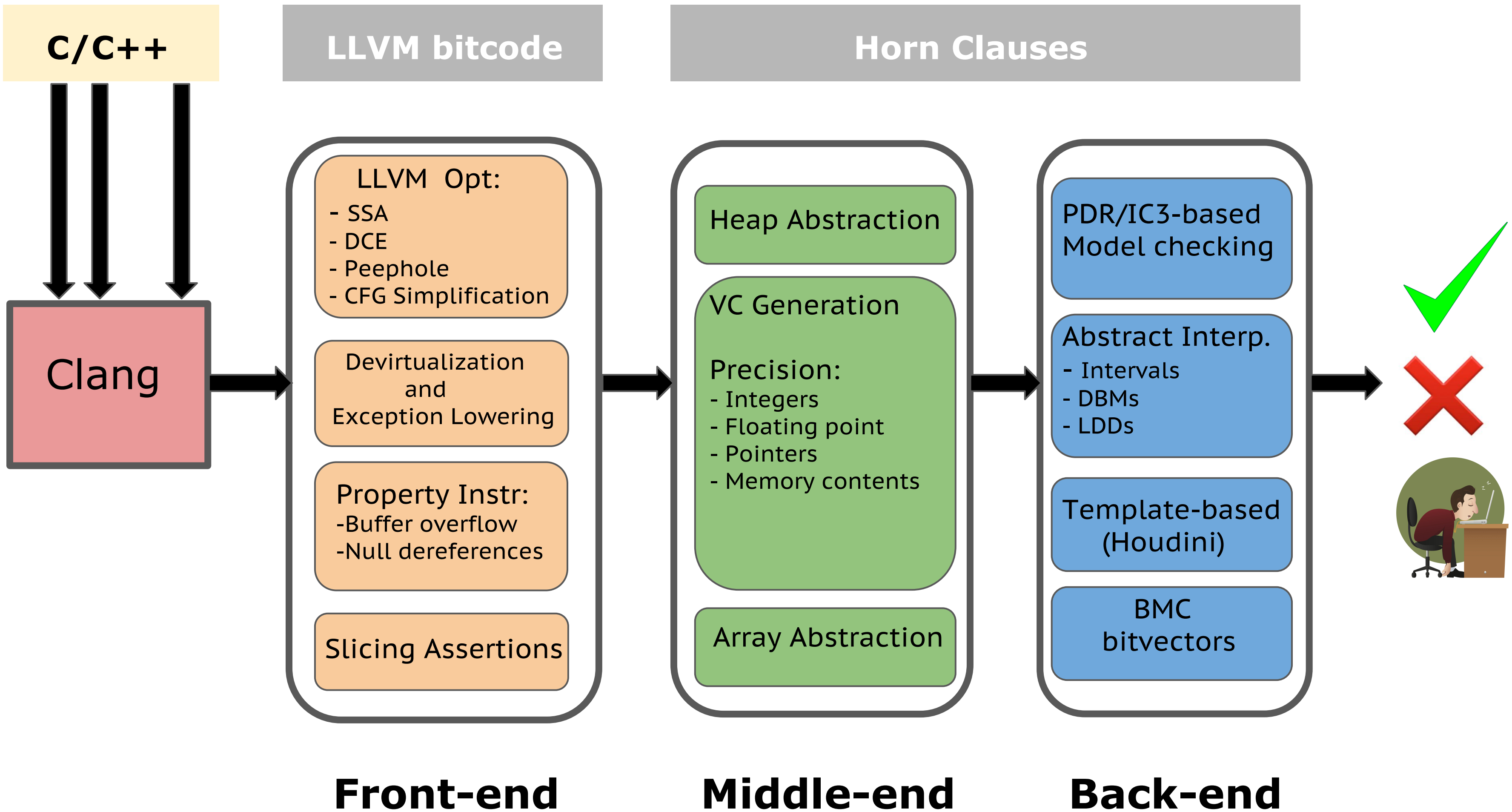
Developed by a domain expert

- understanding of verification techniques is useful (but not required)
- 3-6 month effort for a new property
  - but many things can be reused between similar properties
  - e.g., memory safety, null-dereference, taint checking, use-after-free, etc.

SeaHorn property checkers:

- memory safety (out of bound uses, null pointer)
  - ongoing work to improve scalability and usability
- taint analysis (being developed by Princeton, see CAV 2018)

Architecture of Seahorn



DEMO

UNIVERSITY OF WATERLOO

Types of Software Model Checking

Bounded Model Checking (BMC)

- look for bugs (bad executions) up to a fixed bound
- usually bound depth of loops and depth of recursive calls
- reduce the problem to SAT/SMT

Predicate Abstraction with CounterExample Guided Abstraction Refinement (CEGAR)

- Construct finite-state abstraction of a program
- Analyze using finite-state Model Checking techniques
- Automatically improve / refine abstraction until the analysis is conclusive

Interpolation-based Model Checking (IMC)

- Iteratively apply BMC with increasing bound
- Generalize from bounded-safety proofs
- reduce the problem to many SAT/SMT queries and generalize from SAT/SMT reasoning

SYMBOLIC MODEL CHECKING

Symbolic model checking

Model is represented symbolically using Boolean formulas

Model checking is performed on the symbolic representation **directly**

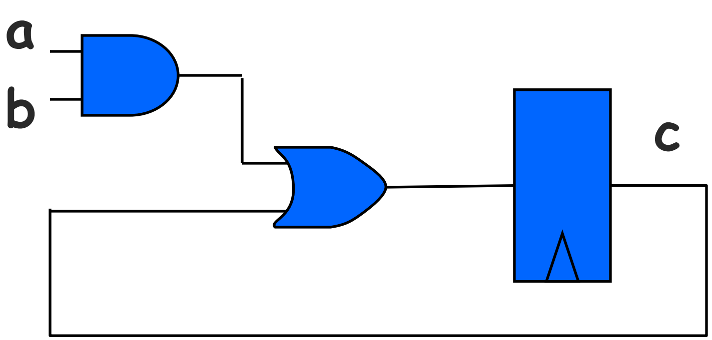
BDD-based

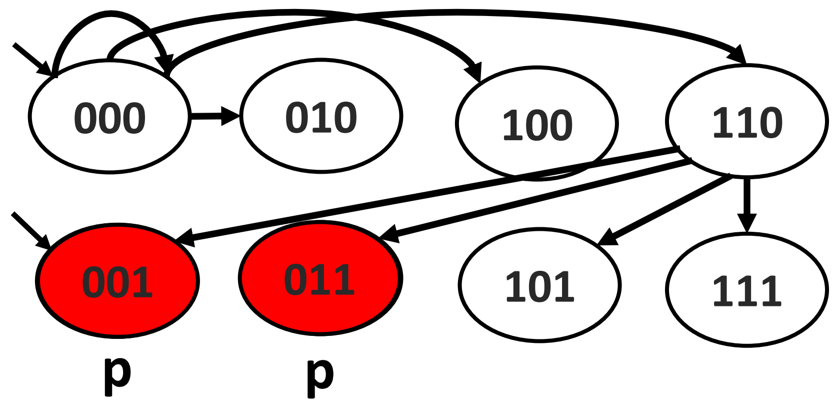
- Use specialized data structure, Binary Decision Diagrams, to represent and manipulate sets of states

SAT-based

- Represent sets of executions using Boolean formulas in Conjunctive Normal Form (CNF)
- Use efficient SAT(isfiability)-solvers for reasoning

Modeling with Propositional Formulas





System is modeled as (V, INIT, T):

- **V** – finite set of Boolean **variables**
  state = valuation to variables

  V = {a, b, c}
  → 8 states: 000,001,…

- **INIT(V)** – describes the set of initial states

  INIT = ¬a ∧ ¬b

- **T(V,V')** – describes the set of transitions

  T = (c' ↔ (a ∧ b) ∨ c)

Atomic Propositions:

- **p(V)** - describes the set of states satisfying p

  p = ¬a ∧ c

Representing Sets as Prop. Formulas

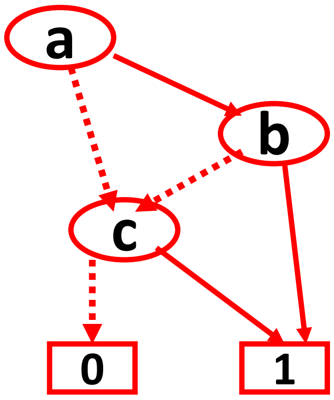
$[F]$ states satisfying F, i.e. $\{\sigma \mid \sigma \vDash F\}$	F propositional formula over V
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	$F_1 \Rightarrow F_2$ i.e. $F_1 \wedge \neg F_2$ unsatisfiable

BDD-based model checking

[J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, LICS'90]

Binary Decision Diagrams (BDDs) are used to represent the **transition relation** and **sets of states**.

can handle systems with **hundreds** of Boolean variables.



ROBDD

Binary decision diagrams (BDDs)

[Bryant, 1986]

Data structure for representing
Boolean functions (propositional formulas)

Often **concise** in memory

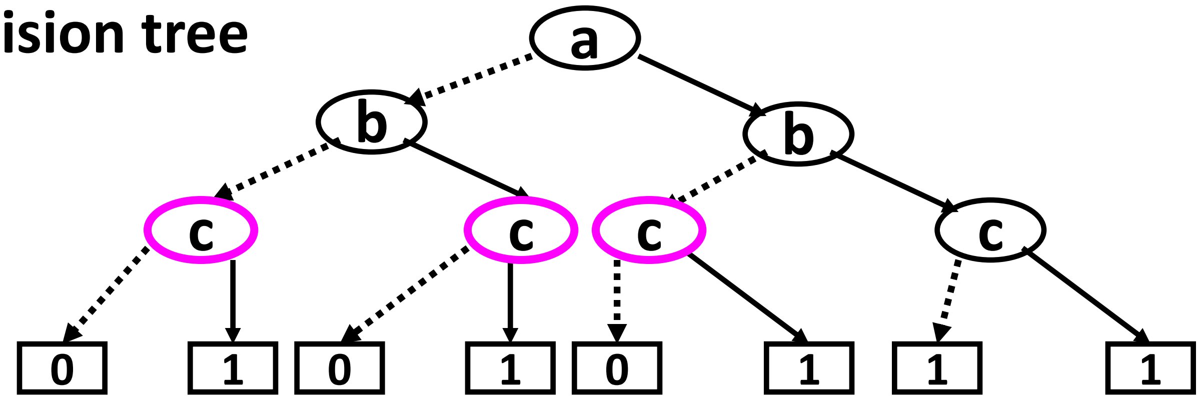
Canonical representation

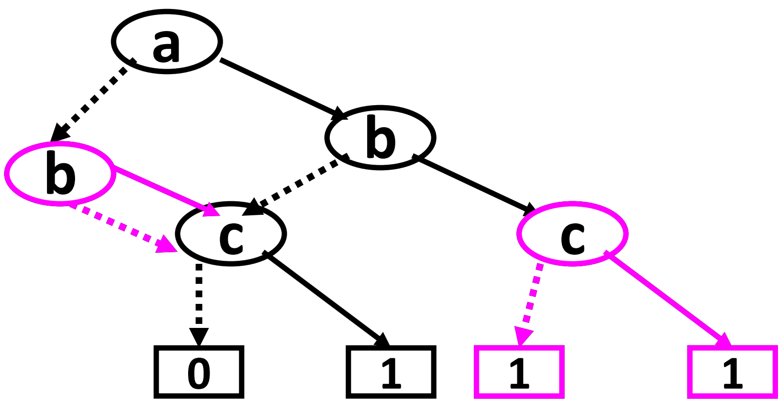
Most **Boolean operations** can be performed on BDDs in **polynomial time** in the BDD size

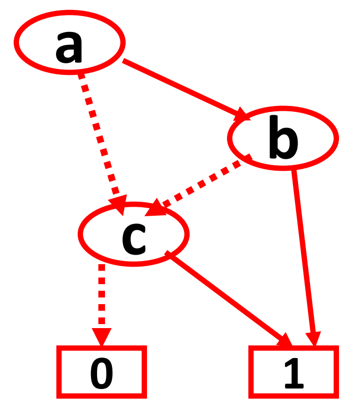
UNIVERSITY OF WATERLOO

BDD for f(a,b,c) = (a ∧ b) ∨ c

Decision tree





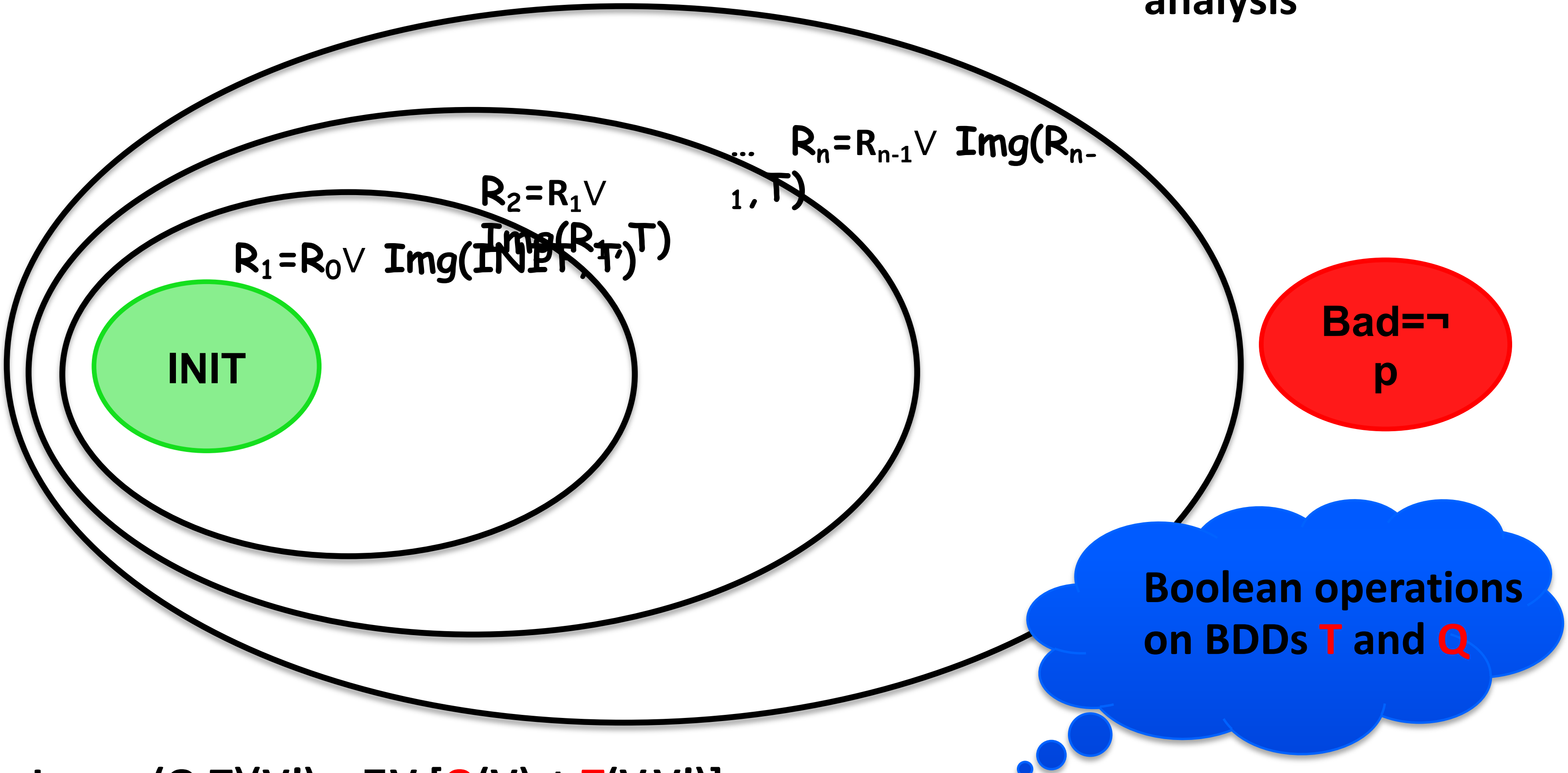


ROBDD

Forward Reachability Analysis with BDDs

Does AG p hold?

All safety properties reduce to reachability analysis



Image(Q,T)(V’) = ∃V [Q(V) ∧ T(V,V’)]

UNIVERSITY OF WATERLOO

Boolean Satisfiability (CNF-SAT)

Let V be a set of variables

A *literal* is either a variable v in V or its negation ~v

A *clause* is a disjunction of literals

- e.g., (v1 || ~v2 || v3)

A Boolean formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

- e.g., (v1 || ~v2) && (v3 || v2)

An *assignment* *s* of Boolean values to variables *satisfies* a clause *c* if it evaluates at least one literal in *c* to true

An assignment *s* *satisfies* a formula *C* in CNF if it satisfies every clause in *C*

Boolean Satisfiability Problem (CNF-SAT):

- determine whether a given CNF C is satisfiable

Algorithms for SAT

SAT is NP-complete

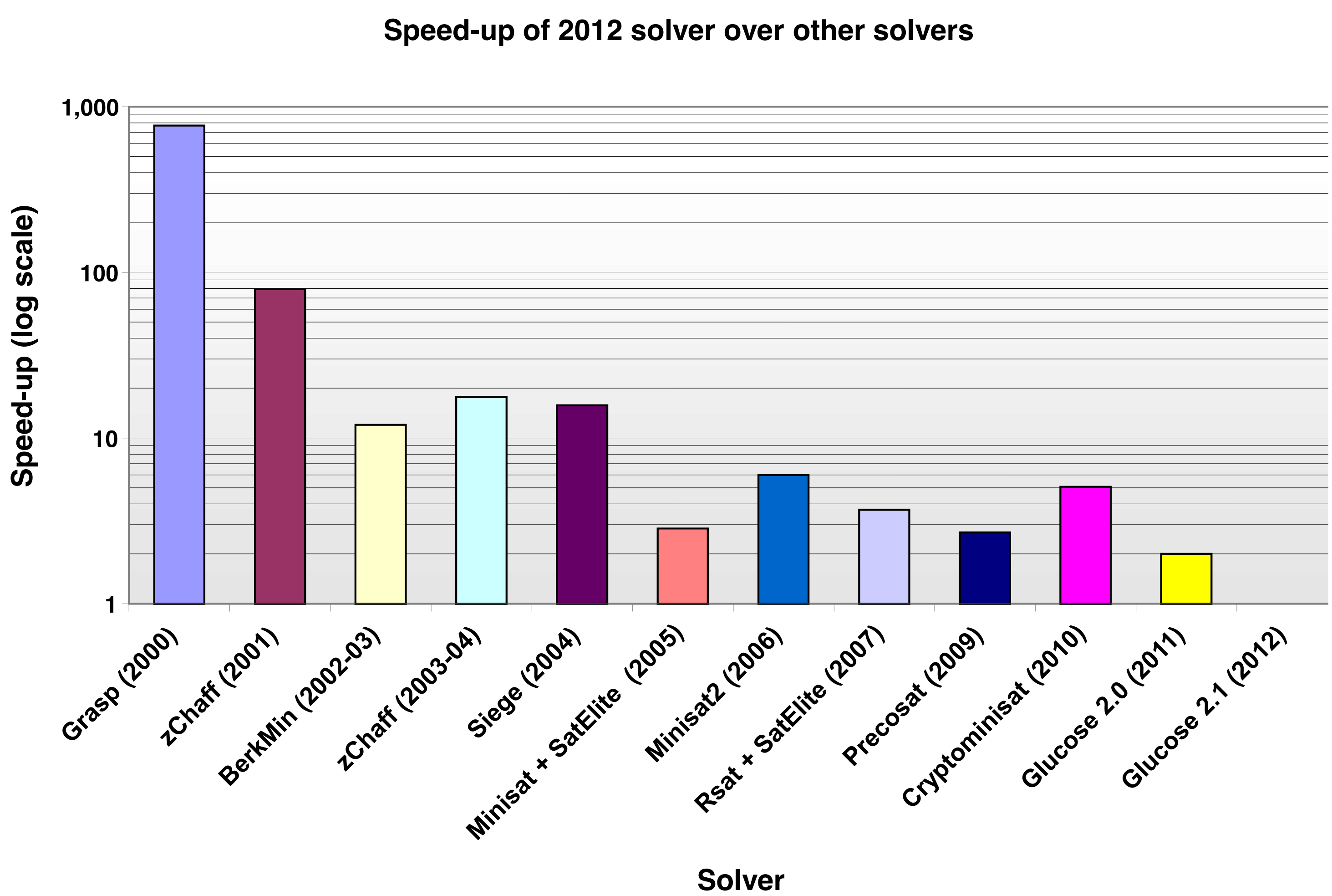
DPLL (Davis-Putnam-Logemman-Loveland, ‘60)

- smart enumeration of all possible SAT assignments
- worst-case EXPTIME
- alternate between deciding and propagating variable assignments

CDCL (GRASP ‘96, Chaff ‘01)

- conflict-driven clause learning
- extends DPLL with
  - smart data structures, backjumping, clause learning, heuristics, restarts…
- scales to millions of variables
- N. Een and N. Sörensson, “An Extensible SAT-solver”, in SAT 2013.

Some Experience with SAT Solving



from M. Vardi, https://www.cs.rice.edu/~vardi/papers/highlights15.pdf

UNIVERSITY OF WATERLOO

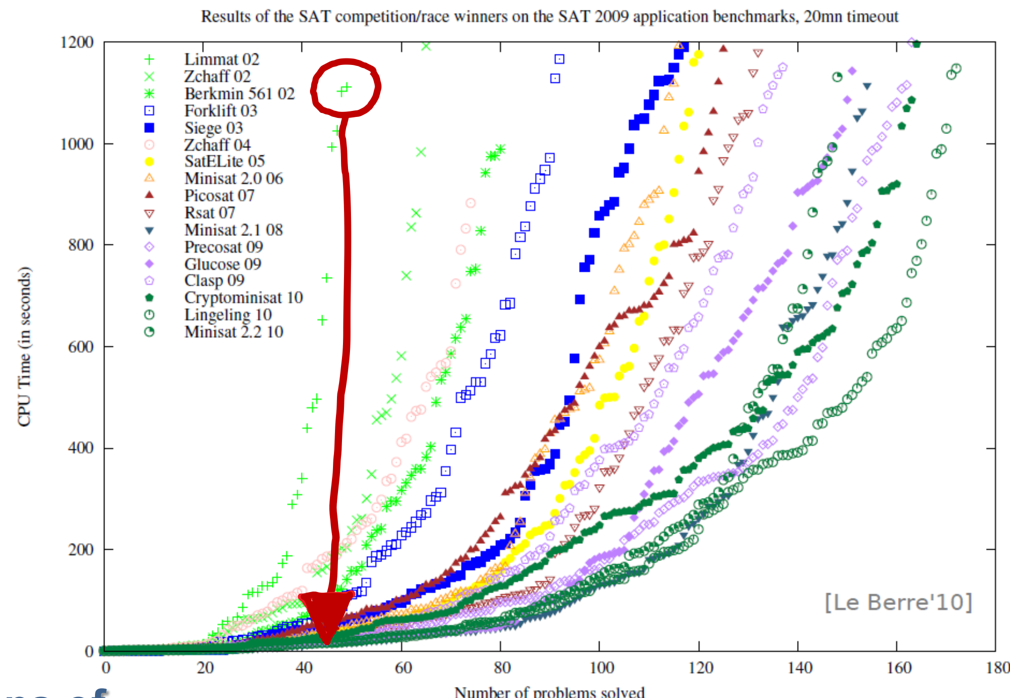
SAT - Milestones

Problems impossible 10 years ago are trivial today

year	Milestone
1960	Davis-Putnam procedure
1962	Davis-Logeman-Loveland
1984	Binary Decision Diagrams
1992	DIMACS SAT challenge
1994	SATO: clause indexing
1997	GRASP: conflict clause learning
1998	Search Restarts
2001	zChaff: 2-watch literal, VSIDS
2005	Preprocessing techniques
2007	Phase caching
2008	Cache optimized indexing
2009	In-processing, clause management
2010	Blocked clause elimination

Concept

2002 **2010**



Millions of variables from HW designs

Courtesy Daniel le Berre

SAT(isfiability)-Solvers

SAT is NP-complete

- but existing tools can solve problems with millions of variables

DPLL (Davis-Putnam-Logemman-Loveland, ‘60)

- smart enumeration of all possible SAT assignments
- worst-case EXPTIME
- alternate between deciding and propagating variable assignments

CDCL (GRASP ‘96, Chaff ’01, MiniSat’03)

- conflict-driven clause learning
- extends DPLL with
  - smart data structures, backjumping, clause learning, heuristics, restarts…
- scales to millions of variables
- N. Een and N. Sörensson, “An Extensible SAT-solver”, in SAT 2013.

SAT-based Model Checking

Bounded Model Checking

- Is there a counterexample of k-steps

Unbounded Model Checking

- Induction and K-Induction (k-IND)
- Interpolation Based Model Checking (IMC)
- Property Directed Reachability (IC3/PDR)

Bounded Model Checking for AG p

Given

- A finite transition system M= (V, I(V), T(V,V'))
- A safety property AG p, where p = p(V)

Determine

- Does M allow a counterexample to p of *k transitions or fewer*?

*** BMC can handle all of LTL formulas**

UNIVERSITY OF WATERLOO

A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, DAC'99

BMC for checking AG p with SAT

Biere, et al. TACAS99

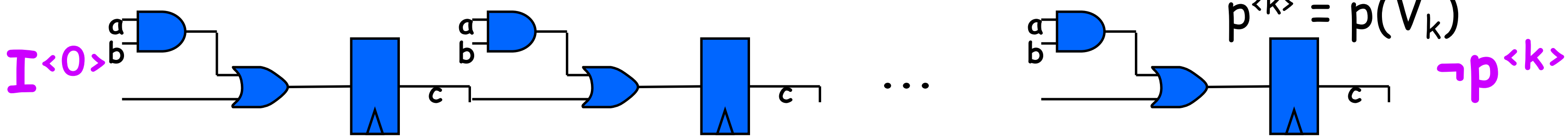
Unfold the model k times:

- $U = T^{<0>} \wedge T^{<1>} \wedge \ldots \wedge T^{<k-1>}$

$I^{<0>} = I(V_0)$

$T^{<i>} = T(V_i, V_{i+1})$

$p^{<k>} = p(V_k)$



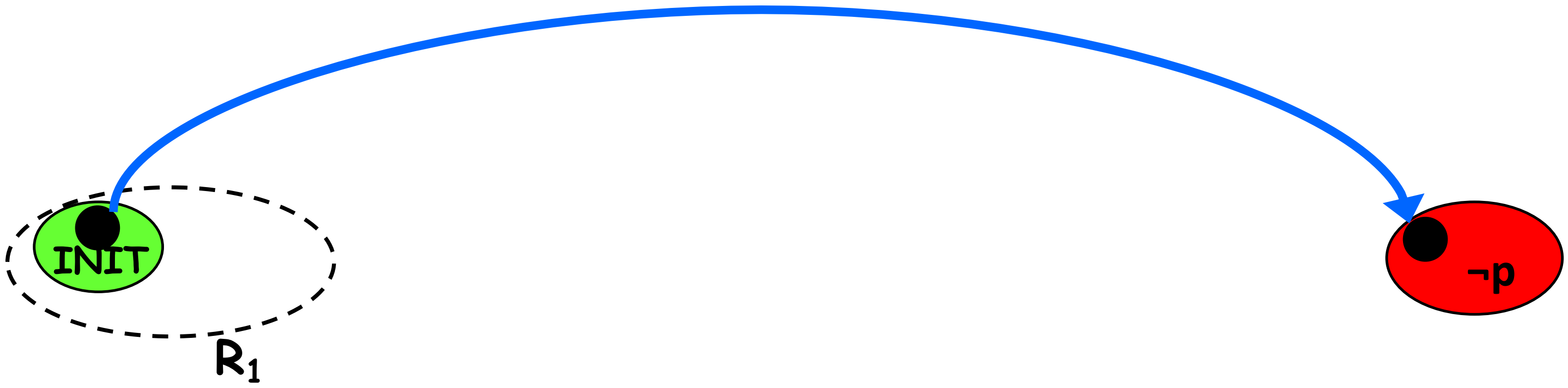
- **Use SAT solver to check satisfiability of**

$$I^{<0>} \wedge U \wedge \neg p^{<k>}$$

- **If satisfiable: the satisfying assignment describes a counterexample of length k**
- **If unsatisfiable: property has no counterexample of length k**

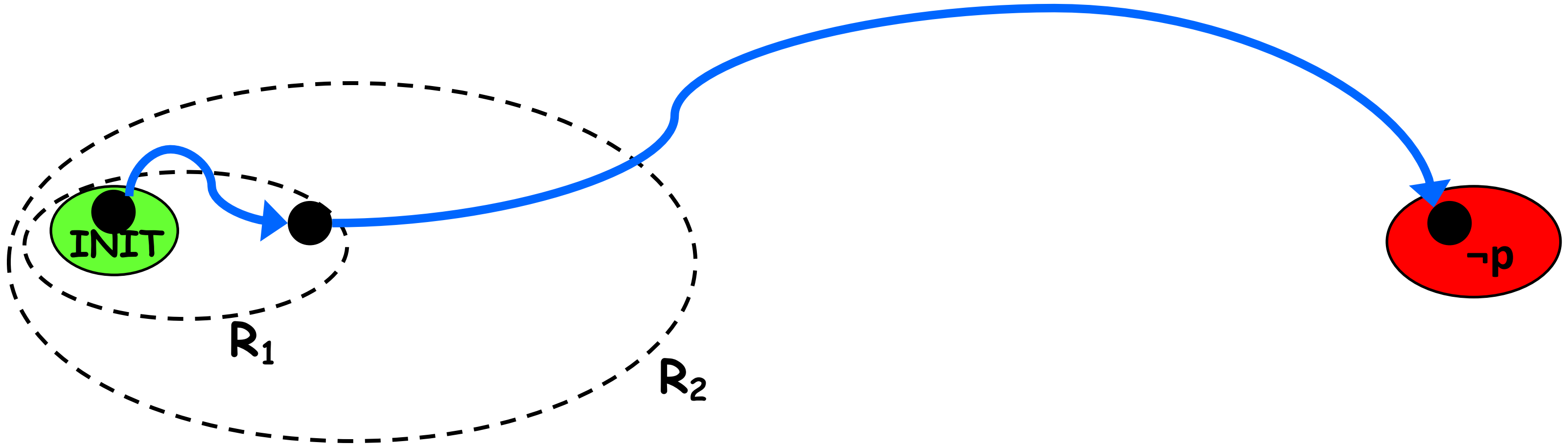
UNIVERSITY OF WATERLOO

Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge \neg p(V^1)$$

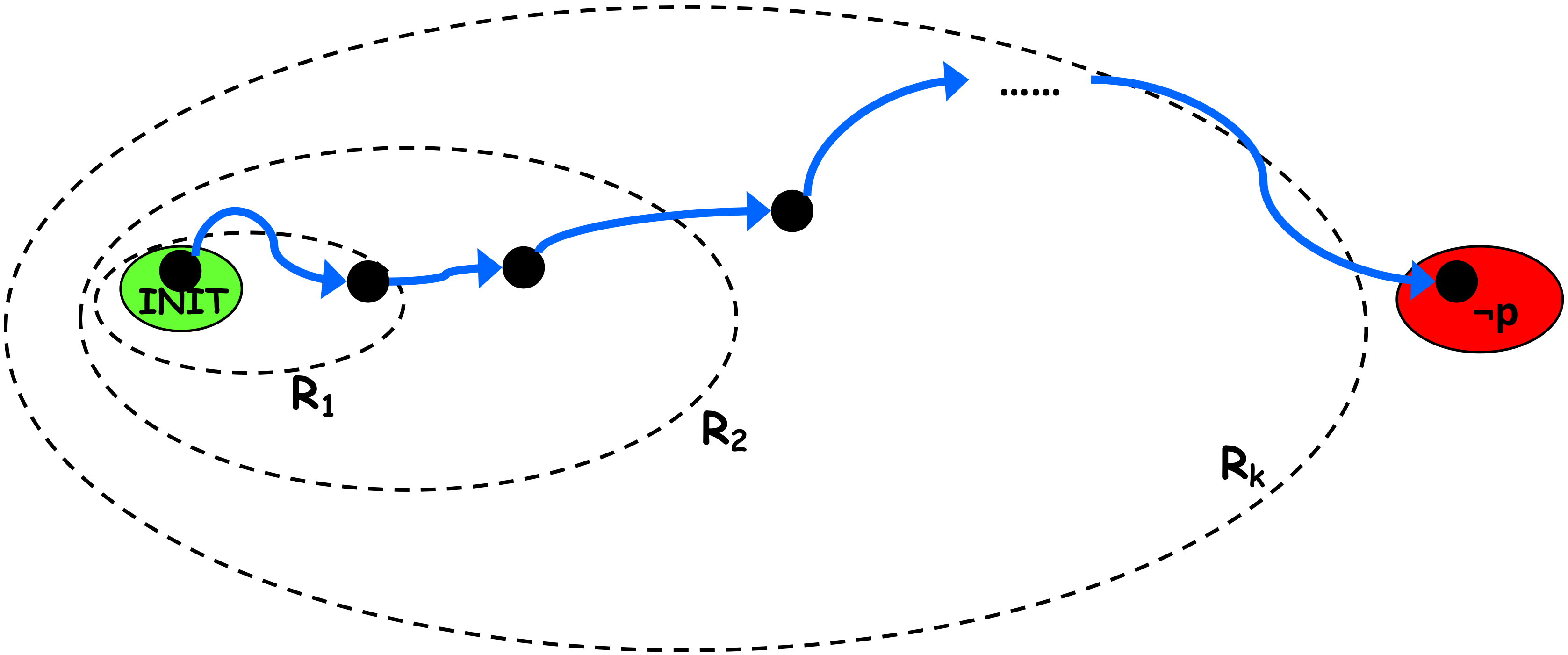
Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge T(V^1,V^2) \wedge \neg p(V^2)$$

UNIVERSITY OF WATERLOO

Bounded Model Checking



$$INIT(V^0) \wedge T(V^0,V^1) \wedge \ldots \wedge T(V^{k-1},V^k) \wedge \neg p(V^k)$$

Bounded Model Checking

Terminates

- with a counterexample or
- with time- or memory-out

=> The method is suitable for **falsification**, not verification

Can be used for **verification** by choosing k which is large enough

- Need bound on length of the shortest counterexample.
  - *diameter* bound. The diameter is the maximum length of the shortest path between any two states.

Using such k is often **not practical**

- Worst case diameter is exponential. Obtaining better bounds is sometimes possible, but generally intractable.

Unbounded SAT-based Model Checking

Induction and K-Induction (k-IND)

Interpolation Based Model Checking (IMC)

Property Directed Reachability (IC3/PDR)

SAT-Based Verification
(unbounded model checking)

Uses BMC for falsification

Simulates forward reachability analysis for verification

Identifies a termination condition

- all reachable states have been found: “fixed-point”

Symbolic Safety and Reachability

A transition system P = (V, Init, Tr, Bad)

P is UNSAFE if and only if there exists a number N s.t.

P is SAFE if and only if there exists a safe inductive invariant Inv s.t.

$$Init(X_0) \wedge \left(\bigwedge_{i=0}^{N-1} Tr(X_i, X_{i+1}) \right) \wedge Bad(X_N) \not\Rightarrow \bot$$

$$Init \Rightarrow Inv$$

$$Inv(X) \wedge Tr(X, X') \Rightarrow Inv(X')$$

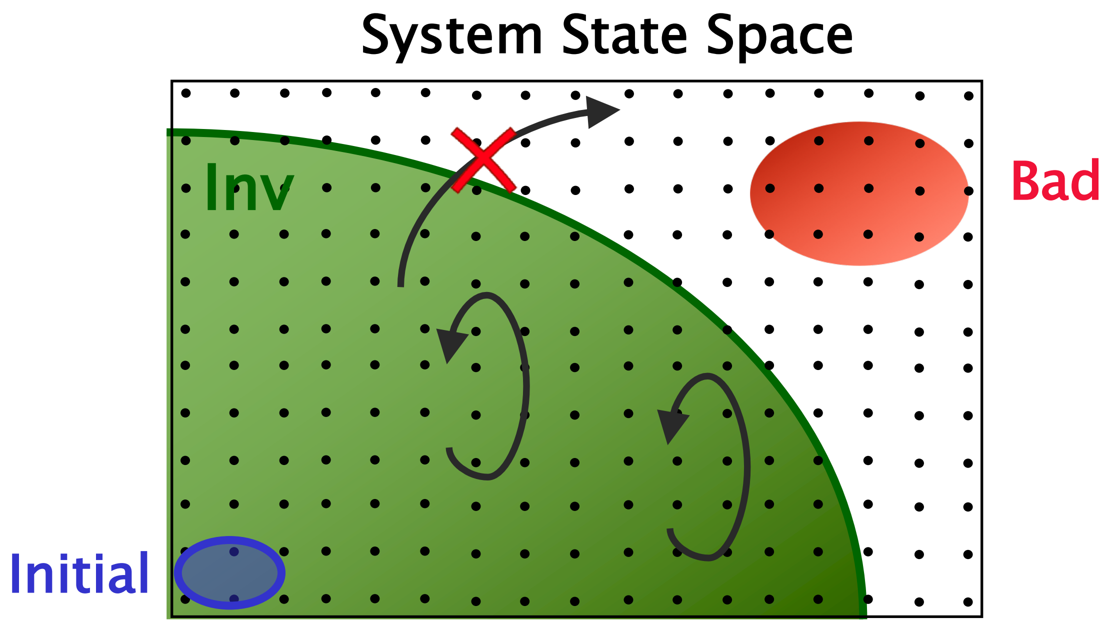
Inductive

$$Inv \Rightarrow \neg Bad$$

Safe

UNIVERSITY OF WATERLOO

Inductive Invariants

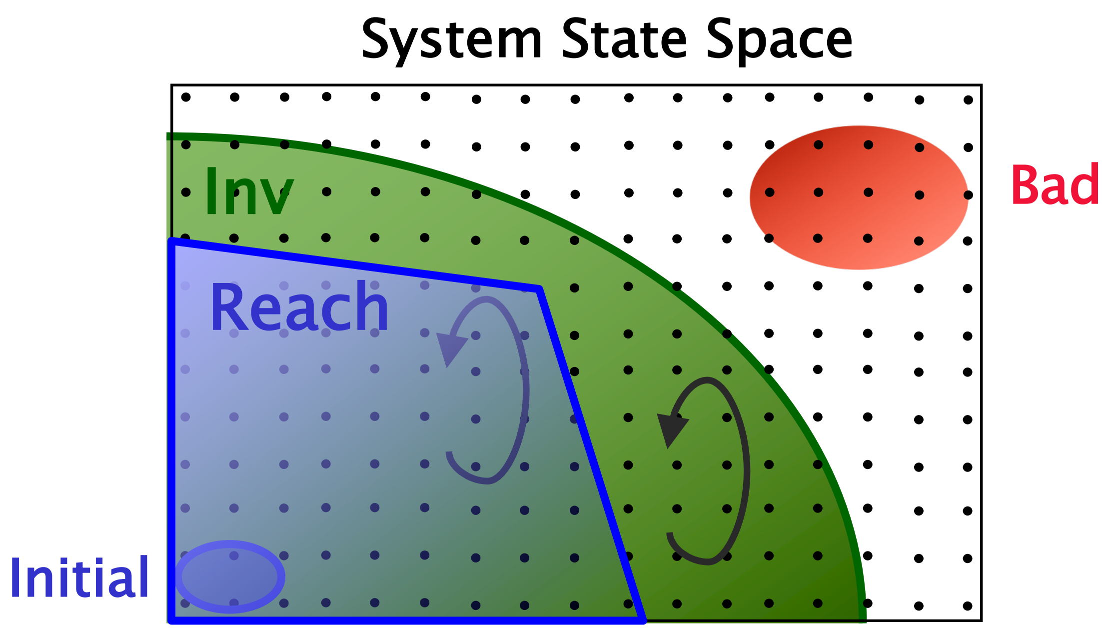


System S is safe iff there exists an inductive invariant **Inv**:

- **Initiation:** Initial ⊆ Inv
- **Safety:** Inv ∩ Bad = ∅
- **Consecution:** TR(Inv) ⊆ Inv

i.e., if s ∈ Inv and s⤳t then t ∈ Inv

Inductive Invariants



System S is safe iff there exists an inductive invariant **Inv**:

- **Initiation:** Initial ⊆ Inv
- **Safety:** Inv ∩ Bad = ∅
- **Consecution:** TR(Inv) ⊆ Inv

i.e., if s ∈ Inv and s⤳t then t ∈ Inv

System S is safe if Reach ∩ Bad = ∅