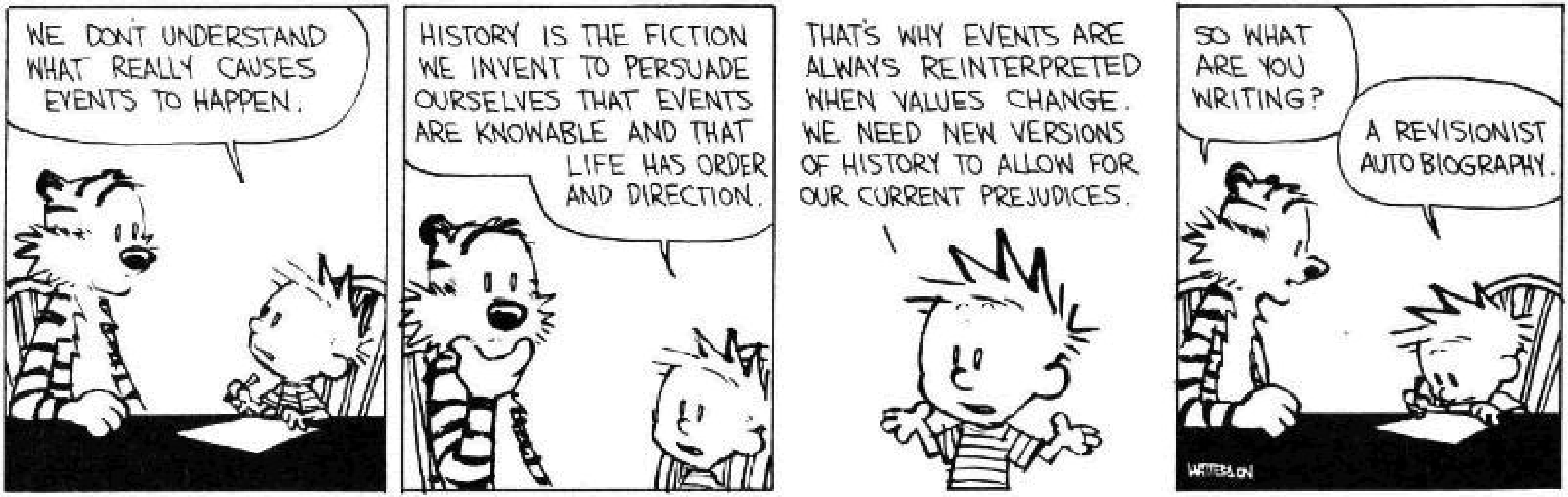
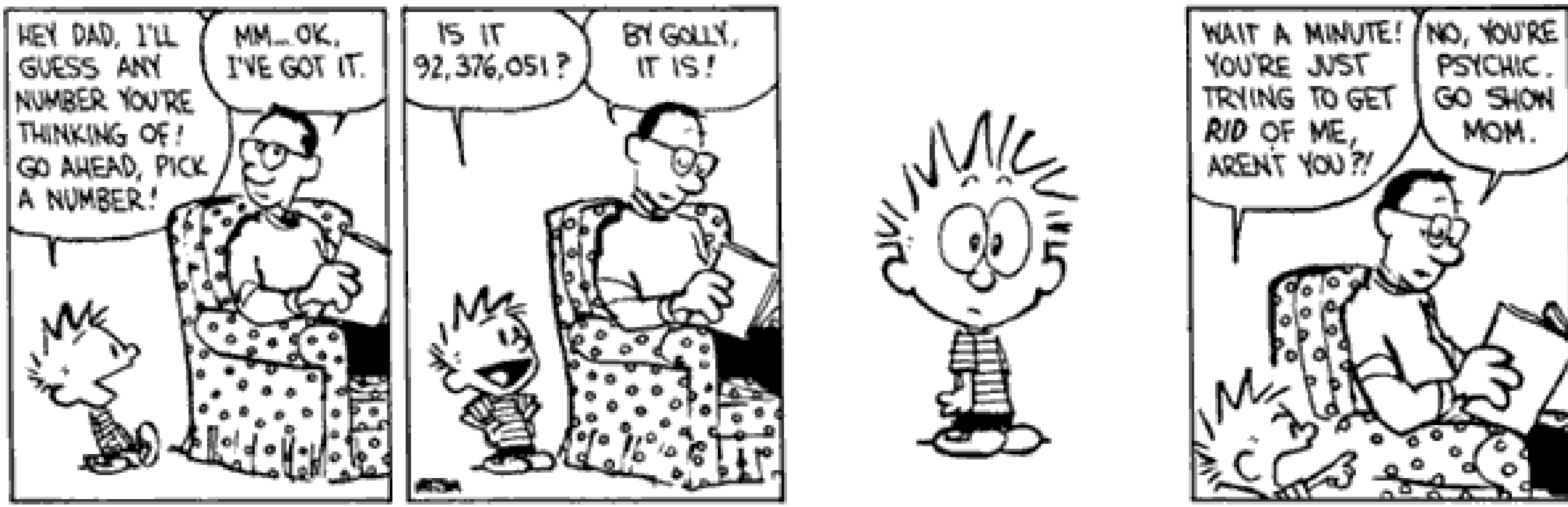


Model Checking

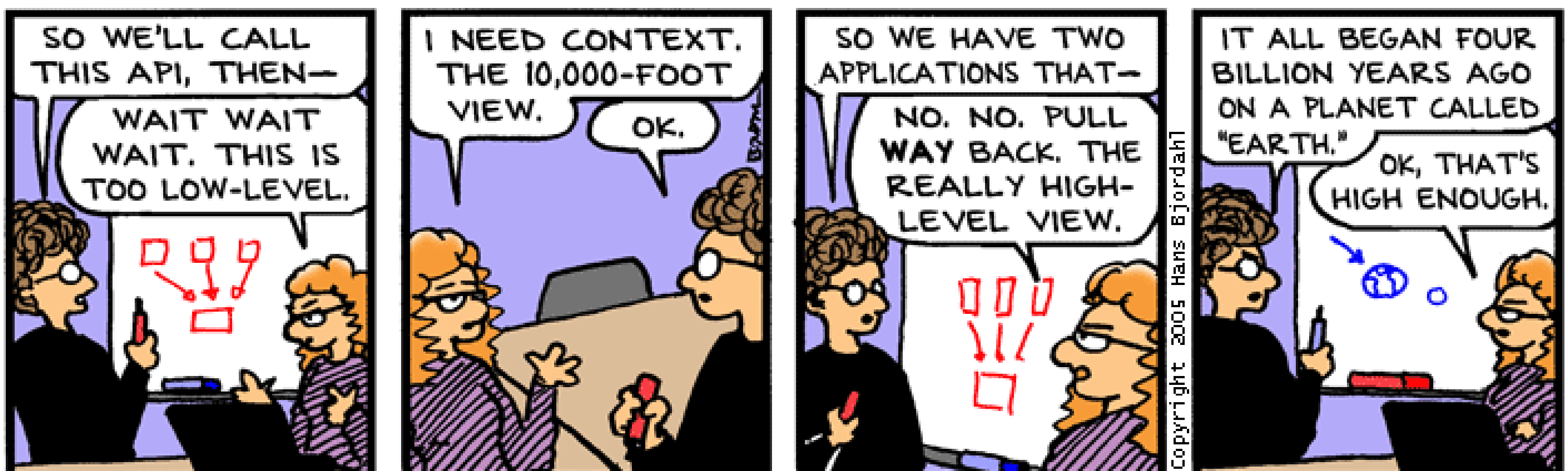


Double Header

- Two Lectures
  - Model Checking
  - Software Model Checking
  - SLAM and BLAST
- “Flying Boxes”
  - It is traditional to describe this stuff (especially SLAM and BLAST) with high-gloss animation.
- Some Key Players:
  - Model Checking: Ed Clarke, Ken McMillan, Amir Pnueli
  - SLAM: Tom Ball, Sriram Rajamani
  - BLAST: Ranjit Jhala, Rupak Majumdar, Tom Henzinger

Who are we again?

- We're going to find critical bugs in important bits of software
  - using PL techniques!
- You will be enthusiastic about this
  - and thus want to learn the gritty details



Bug Bash by Hans Bjordahl http://www.bugbash.net/

Take-Home Message

- Model checking is the exhaustive exploration of the state space of a system, typically to see if an error state is reachable. It produces concrete counter-examples.
- The state explosion problem refers to the large number of states in the model.
- Temporal logic allows you to specify properties with concepts like “eventually” and “always”.

Overarching Plan

- Model Checking *(Today)*
  - Transition Systems (Models)
  - Temporal Properties
  - LTL and CTL
  - (Explicit State) Model Checking
  - Symbolic Model Checking
- Counterexample Guided Abstraction Refinement
  - Safety Properties
  - Predicate Abstraction (“c2bp”)
  - Software Model Checking (“bebop”)
  - Counterexample Feasibility (“newton”, “hw 5”)
  - Abstraction Refinement (weakest pre, thrm prvr)

Spoiler Space

- This stuff really works!
  - This is not ESC or PCC or Denotational Semantics
- Symbolic Model Checking is a massive success in the model-checking field
  - I know people who think Ken McMillan walks on water in a “ha-ha-ha only serious” way
- SLAM took the PL world by storm
  - Spawned multiple copycat projects
  - Incorporated into Windows DDK as “static driver verifier”

Topic: (Generic) Model Checking

- There are complete courses in model checking; I will skim.
  - *Model Checking* by Edmund C. Clarke, Orna Grumberg, and Doron A. Peled, MIT press
  - *Symbolic Model Checking* by Ken McMillan

Model Checking

- Model checking is an *automated* technique
- Model checking verifies *transition systems*
- Model checking verifies *temporal properties*
- Model checking can be also used for falsification by generating *counter-examples*
- Model Checker: A program that checks if a (transition) system satisfies a (temporal) property

Verification vs. Falsification

- An automated verification tool
  - can report that the system is verified (with a **proof**)
  - or that the system was not verified (with **???**)
- When the system was not verified it would be helpful to explain why
  - Model checkers can output an error counter-example: a concrete execution scenario that demonstrates the error
- Can view a model checker as a falsification tool
  - The main goal is to find bugs
- OK, so what can we verify or falsify?

Temporal Properties

- Temporal Property: A property with time-related operators such as "invariant" or "eventually"
- Invariant(*p*): is true in a state if property *p* is true in every state on all execution paths starting at that state
  - The Invariant operator has different names in different temporal logics:
    - **G, AG, □ ("goal" or "box" or "forall")**
- Eventually(*p*): is true in a state if property *p* is true at some state on every execution path starting from that state
    - **F, AF, ◇ ("diamond" or "future" or "exists")**

An Example Concurrent Program

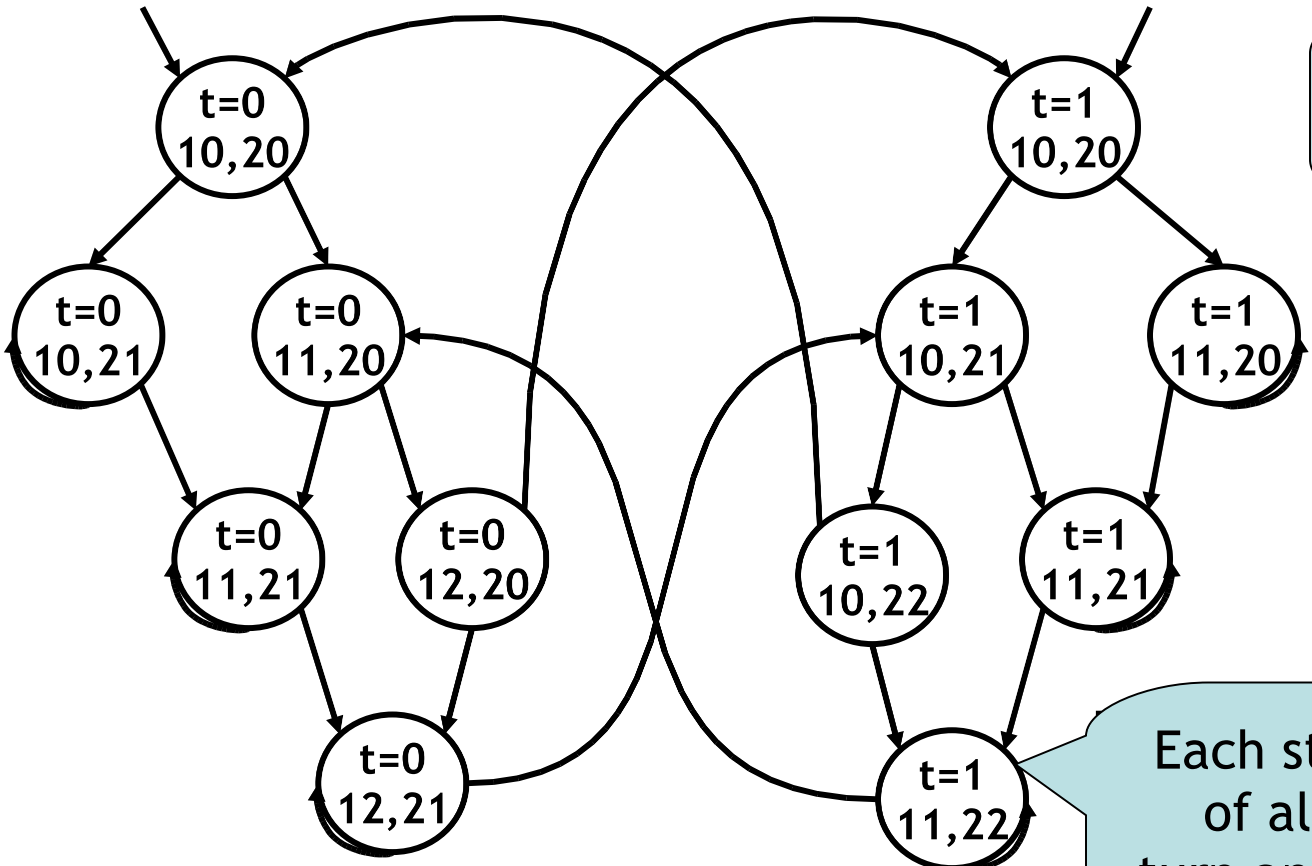
- A simple concurrent mutual exclusion program
- Two processes execute asynchronously
- There is a shared variable **`turn`**
- Two processes use the shared variable to ensure that they are not in the critical section at the same time
- Can be viewed as a "fundamental" program: any bigger concurrent one would include this one

```
10: while True do
11:     wait(turn = 0);
    // critical section
12:     turn := 1;
13:   end while;

|| // concurrently with

20: while True do
21:    wait(turn = 1);
     // critical section
22:    turn := 0;
23: end while
```

Reachable States of the Example Program



Next: formalize this intuition …

Each state is a valuation of all the variables: turn and the two program counters for two processes

Transition Systems

- In model checking the system being analyzed is represented as a labeled transition system

$$T = (S, I, R, L)$$

  - Also called a Kripke Structure
  - S = Set of states // standard FSM
  - $I \subseteq S$ = Set of initial states // standard FSM
  - $R \subseteq S \times S$ = Transition relation // standard FSM
  - $L: S \rightarrow \mathcal{P}(AP)$ = Labeling function // this is new!
- *AP*: Set of atomic propositions (e.g., "x=5"$\in$AP)
  - Atomic propositions capture basic properties
  - For software, atomic props depend on variable values
  - The labeling function labels each state with the set of propositions true in that state

Properties of the Program

- Example: "In all the reachable states (configurations) of the system, the two processes are *never in the critical section at the same time*"
  - Equivalently, we can say that
    - *Invariant*($\neg$(pc1=12 $\wedge$ pc2=22))
- Also: "*Eventually the first process enters* the critical section"
    - *Eventually*(pc1=12)
- "pc1=12", "pc2=22" are atomic properties

Temporal Logics

- There are four basic temporal operators:
- *X p* = Ne<u>x</u>t p, p holds in the next state
- *G p* = <u>G</u>lobally p, p holds in every state, p is an invariant
- *F p* = <u>F</u>uture p, p will hold in a future state, p holds eventually
- *p U q* = p <u>U</u>ntil q, assertion p will hold until q holds
- Precise meaning of these temporal operators are defined on execution paths

Execution Paths

- A path in a transition system is an infinite sequence of states

  $(s_0, s_1, s_2, \ldots)$, such that $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$
- A path $(s_0, s_1, s_2, \ldots)$ is an execution path if $s_0 \in I$
- Given a path $x = (s_0, s_1, s_2, \ldots)$
  - x_i denotes the i^{th} state s_i
  - x^i denotes the i^{th} suffix $(s_i, s_{i+1}, s_{i+2}, \ldots)$

- In some temporal logics one can quantify the paths starting from a state using path quantifiers
  - A : for all paths
  - E : there exists a path

Linear Time Logic (LTL)

- LTL properties are constructed from atomic propositions in AP; logical operators $\wedge$, $\vee$, $\neg$; and temporal operators X, G, F, U.
- The semantics of LTL properties is defined on paths:

Given a path x:

$x \vDash p$	iff	$L(x_0, p)$	// atomic prop
$x \vDash X\ p$	iff	$x^1 \vDash p$	// next
$x \vDash F\ p$	iff	$\exists i \geq 0.\ x^i \vDash p$	// future
$x \vDash G\ p$	iff	$\forall i \geq 0.\ x^i \vDash p$	// globally
$x \vDash p\ U\ q$	iff	$\exists i \geq 0.\ x^i \vDash q$ and $\forall j < i.\ x^j \vDash p$	// until

Satisfying Linear Time Logic

- Given a transition system $T = (S, I, R, L)$ and an LTL property p, T satisfies p if all paths starting from all initial states I satisfy p
- Example LTL formulas:
  - *Invariant*$(\neg(pc1=12 \wedge pc2=22))$:

    $$G(\neg(pc1=12 \wedge pc2=22))$$

  - *Eventually*$(pc1=12)$:

    $$F(pc1=12)$$

Computation Tree Logic (CTL)

- In CTL temporal properties use path quantifiers
  - A : for all paths
  - E : there exists a path
- The semantics of CTL properties is defined on states:

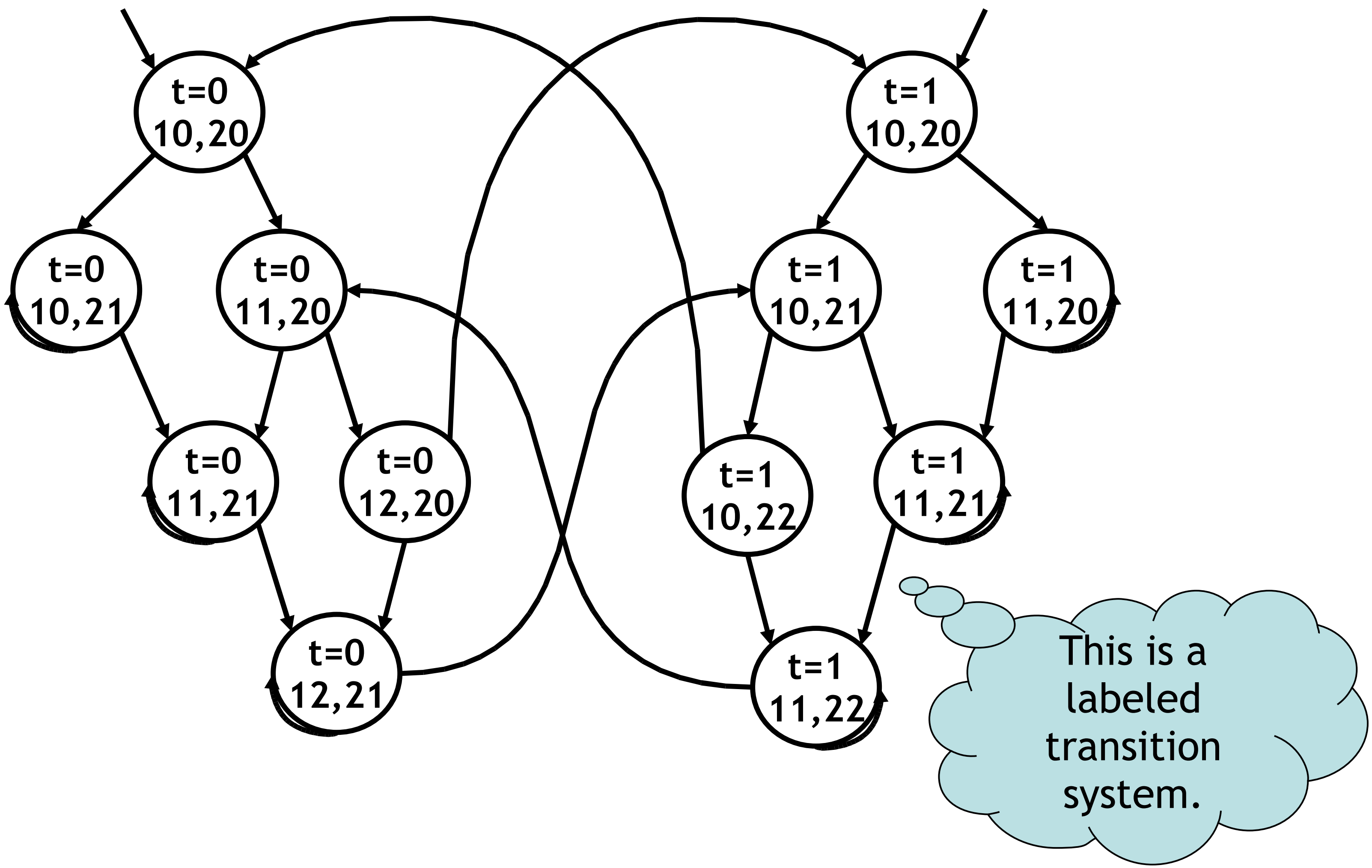
Given a path x

$s \models p$	iff	$L(s, p)$
$s_0 \models EX\ p$	iff	$\exists$ a path $(s_0, s_1, s_2, \ldots)$. $s_1 \models p$
$s_0 \models AX\ p$	iff	$\forall$ paths $(s_0, s_1, s_2, \ldots)$. $s_1 \models p$
$s_0 \models EG\ p$	iff	$\exists$ a path $(s_0, s_1, s_2, \ldots)$. $\forall i \geq 0$. $s_i \models p$
$s_0 \models AG\ p$	iff	$\forall$ paths $(s_0, s_1, s_2, \ldots)$. $\forall i \geq 0$. $s_i \models p$

Linear vs. Branching Time

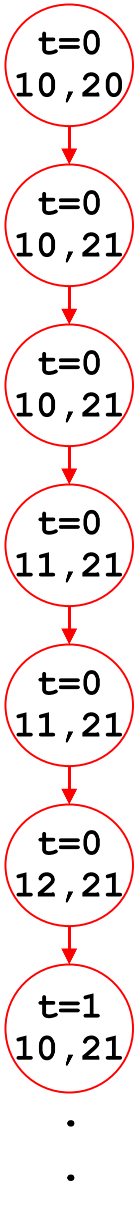
- LTL is a linear time logic
  - When determining if a path satisfies an LTL formula we are only concerned with a single path
- CTL is a branching time logic
  - When determining if a state satisfies a CTL formula we are concerned with multiple paths
  - In CTL the computation is not viewed as a single path but as a computation tree which contains all the paths
  - The computation tree is obtained by unrolling the transition relation
- The expressive powers of CTL and LTL are incomparable (LTL $\subseteq$ CTL*, CTL $\subseteq$ CTL*)
  - Basic temporal properties can be expressed in both logics
  - Not in this lecture, sorry! (Take a class on Modal Logics)

Remember the Example



Linear vs. Branching Time

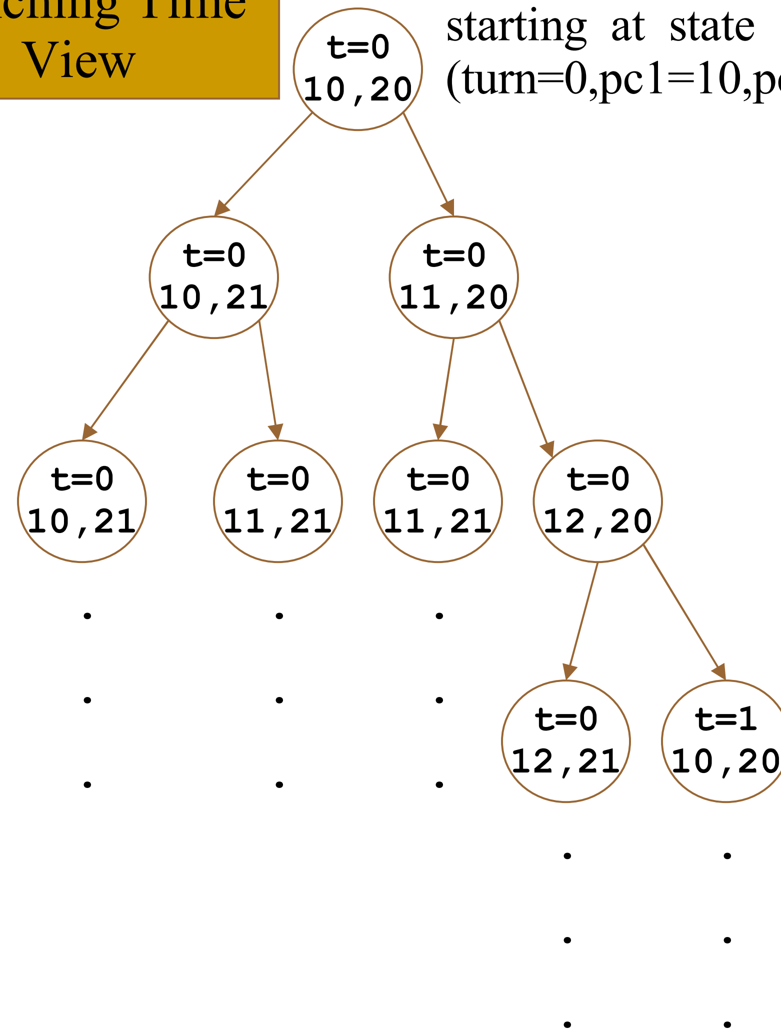
One path starting at state (turn=0,pc1=10,pc2=20)



Linear Time View

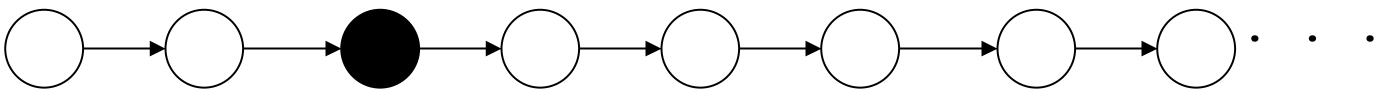
Branching Time View

A computation tree starting at state (turn=0,pc1=10,pc2=20)

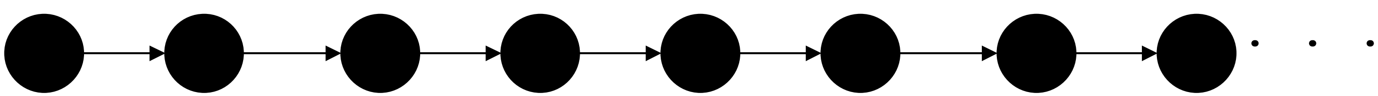


LTL Satisfiability Examples

p does not hold   p holds



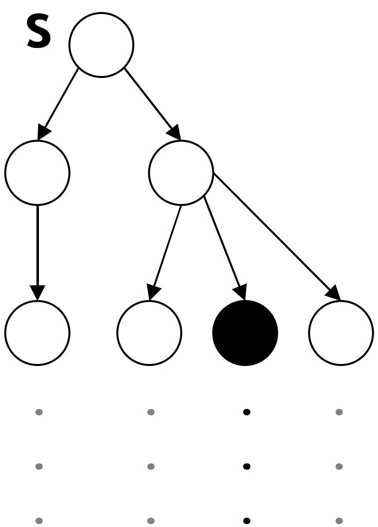
On this path: F p holds, G p does not hold, p does not hold, X p does not hold, X (X p) holds, X (X (X p)) does not hold



On this path: F p holds, G p holds, p holds, X p holds, X (X p) holds, X (X (X p))) holds p does not hold

p holds

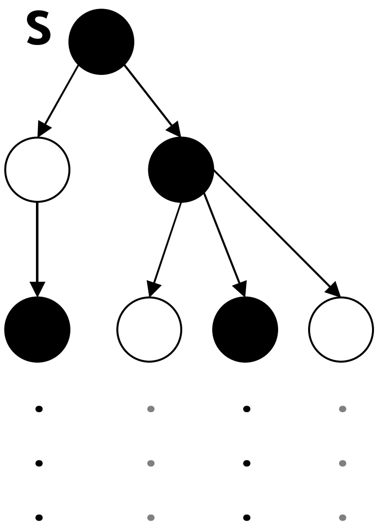
CTL Examples



At state s:

EF p, EX (EX p),
AF (¬p), ¬p holds

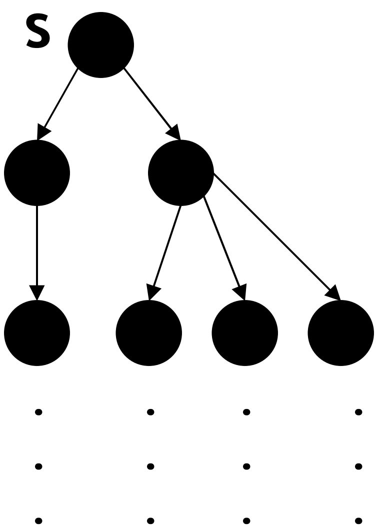
AF p, AG p,
AG (¬p), EX p,
EG p, p does not hold



At state s:

EF p, AF p,
EX (EX p),
EX p, EG p, p holds

AG p, AG (¬p),
AF (¬p) does not hold



At state s:

EF p, AF p,
AG p, EG p,
Ex p, AX p, p holds

EG (¬ p), EF (¬p),
does not hold

Model Checking Complexity

- Given a transition system T = (S, I, R, L) and a CTL formula f
  - One can check if a state of the transition system satisfies the temporal logic formula f in $O(|f| \times (|S| + |R|))$ time
- Given a transition system T = (S, I, R, L) and an LTL formula f
  - One can check if the transition system satisfies the temporal logic formula f in $O(2^{|f|} \times (|S| + |R|))$ time
- Model checking procedures can generate counter-examples without increasing the complexity of verification (= "for free")

Which is slower?

State Space Explosion

- The complexity of model checking increases linearly with respect to the size of the transition system (|S| + |R|)
- However, the size of the transition system (|S| + |R|) is ***exponential*** in the number of variables and number of concurrent processes
- This exponential increase in the state space is called the state space explosion
  - Dealing with it is one of the major challenges in model checking research

Explicit-State Model Checking

- One can show the complexity results using depth first search algorithms
  - The transition system is a directed graph
  - CTL model checking is multiple depth first searches (one for each temporal operator)
  - LTL model checking is one nested depth first search (i.e., two interleaved depth-first-searches)
  - Such algorithms are called explicit-state model checking algorithms *(details on next slides)*

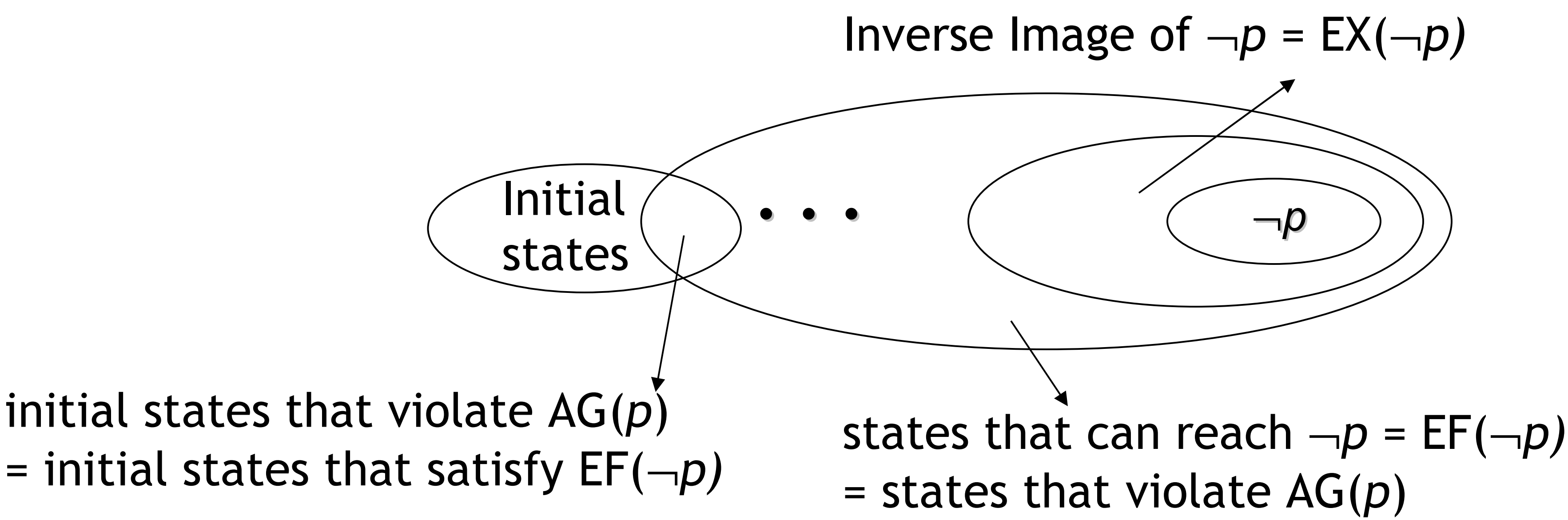
Temporal Properties ≡ Fixpoints

- States that satisfy AG(p) are all the states which are not in EF(¬p) (= the states that can reach ¬p)
- Compute EF(¬p) as the **fixpoint** of Func: $2^S \rightarrow 2^S$
- Given $Z \subseteq S$,
  - Func(Z) = ¬p ∪ **reach-in-one-step**(Z)
  - or Func(Z) = ¬p ∪ EX(Z)

This is called the ***inverse image*** *of Z*

- Actually, EF(¬p) is the ***least*-fixpoint** of Func
  - smallest set Z such that Z = Func(Z)
  - to compute the least fixpoint, start the iteration from Z= ∅, and apply the Func until you reach a fixpoint
  - This can be computed (unlike most other fixpoints)

Pictoral Backward Fixpoint



This fixpoint computation can be used for:

- verification of EF($\neg$p)
- or falsification of AG(p)

… and a similar forward fixpoint handles the other cases

Symbolic Model Checking

- Symbolic Model Checking represent state sets and the transition relation as *Boolean logic formulas*
  - Fixpoint computations manipulate sets of states rather than individual states
  - Recall: we needed to compute EX(Z), but $Z \subseteq S$
- Forward and backward fixpoints can be computed by iteratively manipulating these formulas
  - Forward, inverse image: Existential variable elimination
  - Conjunction (intersection), disjunction (union) and negation (set difference), and equivalence check
- Use an efficient data structure for manipulation of Boolean logic formulas
  - Binary Decision Diagrams (BDDs)

Binary Decision Diagrams (BDDs)

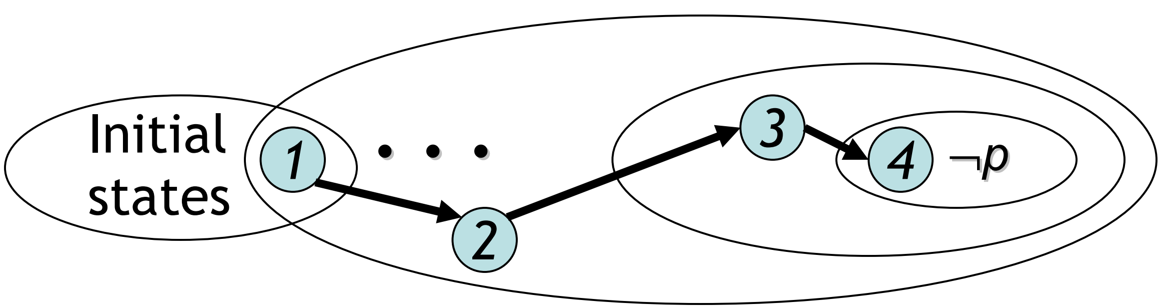
- Efficient representation for boolean functions (a set can be viewed as a function)
- Disjunction, conjunction complexity: at most quadratic
- Negation complexity: constant
- Equivalence checking complexity: constant or linear
- Image computation complexity: can be exponential

Symbolic Model Checking Using BDDs

- SMV (Symbolic Model Verifier) was the first CTL model checker to use a BDD representation
- It has been successfully used in verification
  - of hardware specifications, software specifications, protocols, etc.
- SMV verifies finite state systems
  - It supports both synchronous and asynchronous composition
  - It can handle boolean and enumerated variables
  - It can handle bounded integer variables using a binary encoding of the integer variables
    - It is not very efficient in handling integer variables although this can be fixed

Where's the Beef

- To produce the explicit counter-example, use the "onion-ring method"
  - A counter-example is a valid execution path
  - For each Image Ring (= set of states), find a state and link it with the concrete transition relation R
  - Since each Ring is "reached in one step from previous ring" (e.g., Ring#3 = EX(Ring#4)) this works
  - Each state z comes with L(z) so you know what is true at each point (= what the values of variables are)



Building Up To: Software Model Checking via Counter-Example Guided Abstraction Refinement

- There are easily two dozen SLAM/BLAST/MAGIC papers; I will skim.

Key Terms

- CEGAR = Counterexample guided abstraction refinement. A successful software model-checking approach. Sometimes called "Iterative Abstraction Refinement".
- SLAM = The first CEGAR project/tool. Developed at MSR.
- Lazy Abstraction = A CEGAR optimization used in the BLAST tool from Berkeley.
- Other terms: c2bp, bebop, newton, npackets++, MAGIC, flying boxes, etc.

So … what *is* Counterexample Guided Abstraction Refinement?

- Theorem Proving?
- Dataflow Analysis?
- Model Checking?

Verification by Theorem Proving

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5:   unlock ();
     return;
}
```

1. Loop Invariants
2. Logical formula
3. Check Validity

Invariant:

$$\mathit{lock} \wedge \mathit{new} = \mathit{old}$$

$$\vee$$

$$\neg\, \mathit{lock} \wedge \mathit{new} \neq \mathit{old}$$

Verification by Theorem Proving

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5:   unlock ();
     return;
}
```

1. Loop Invariants
2. Logical formula
3. Check Validity

- Loop Invariants
- Multithreaded Programs

\+ Behaviors encoded in logic

\+ Decision Procedures

Precise [ESC, PCC]

Verification by Program Analysis

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5:   unlock ();
     return;
}
```

1. Dataflow Facts
2. Constraint System
3. Solve constraints

- Imprecision due to fixed facts

+ Abstraction

+ Type/Flow Analyses

Scalable [CQUAL, ESP, MC]

Verification by Model Checking

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5:   unlock ();
     return;
}
```

1. (Finite State) Program
2. State Transition Graph
3. Reachability

- Pgm → Finite state model
- State explosion

\+ State Exploration

\+ Counterexamples

Precise [SPIN, SMV, Bandera, JPF]

One Ring To Rule Them All?

Combining Strengths

Theorem Proving

- Need loop invariants
(will find automatically)
+ Behaviors encoded in logic
(used to refine abstraction)
+ Theorem provers
(used to compute successors, refine abstraction)

Program Analysis

- Imprecise
(will be precise)
+ Abstraction
(will shrink the state space we must explore)

SLAM

Model Checking

- Finite-state model, state explosion
(will find small good model)
+ State Space Exploration
(used to get a path sensitive analysis)
+ Counterexamples
(used to find relevant facts, refine abstraction)

Homework

- Read *Lazy Abstraction*
- Optionally read *TAR*