CSC2125: Safety and Certification of Autonomous Vehicles

Lecture 2 (and perhaps a bit of 3) Verification and Testing Primer

Marsha Chechik

Validation of Systems

The principal methods for the validation of complex parallel and distributed systems are:

- Testing (using the system itself)
- Simulation (using a model of the system)
- Deductive verification (mathematical (manual) proof of correctness, in practice done with computer aided proof assistants/theorem provers)
- Model Checking (≈ exhaustive testing of a model of the system)

Simulation

The main method for the validation of hardware designs:

- When designing new microprocessors, no physical silicon implementation exists until very late in the project.
- Example: Intel Pentium 4 simulation capacity (Roope Kaivola, talk at CAV05):
  - 8000 CPUs
  - Full chip simulation speed 8 Hz (final silicon > 2 GHz).
  - Amount of real time simulated before tape-out: around 2 minutes.

Deductive Verification

- Proving things correct by mathematical means (mostly invariants + induction).
- Computer aided proof assistants/theorem provers used to keep you honest and to prove sub-cases.
- Very high cost, requires highly skilled personnel:
  - Only for truly critical systems.
  - HW examples: Pentium 4 FPU, Pentium 4 register rename logic (Roope Kaivola: 2 man years, 2 'time bomb' silicon bugs found - thankfully masked by surrounding logic)

Plan for this portion of the material

- Model-checking intro
  - Temporal logics
  - Models
  - Bounded model-checking with SAT
- Testing coverage and adequacy criteria
- Deductive verification by an example (if time allows)

Material for these slides came from CSC410 (Testing and Verification), CSC2108 (Automated Verification) and an Advanced BMC tutorial by Keijo Heljanko and Tommi Junttila

Model Checking Intro

Guaranteeing system properties

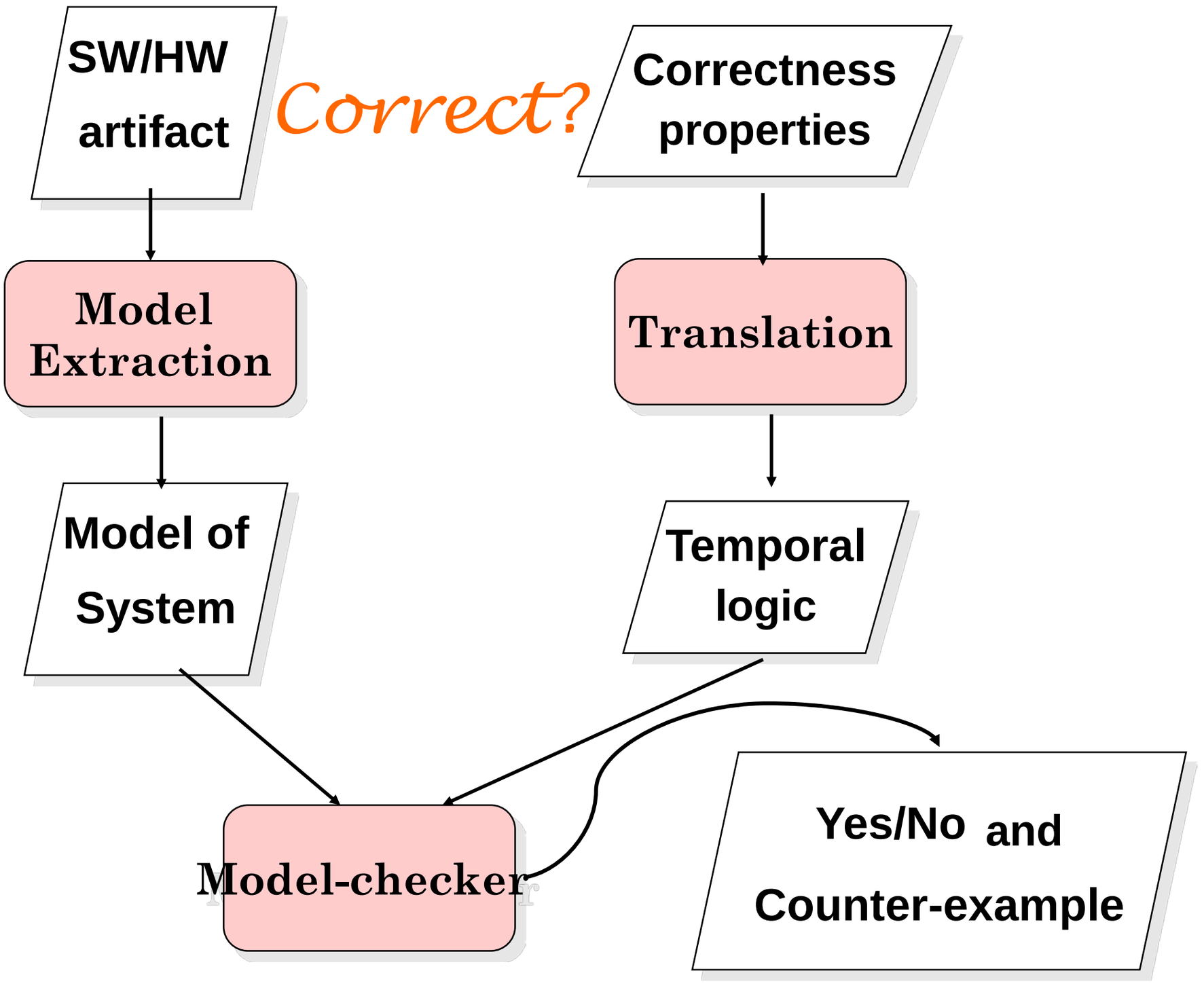
- Safety critical systems:
  - Air traffic control systems.
  - Launch and control systems for space exploration.
  - Nuclear power plants.
- For complex systems like these, certain system properties must be *guaranteed* to hold.
- In other words, certain consequences must follow from the specification of the system.

  e.g., *Two aircrafts must never try to use the same incoming runway within a one minute interval.*

Model checking

- In systems that respond to events or signals, it is often useful to think of the system as starting in some initial state, and making transitions from state to state.
  examples: telephones, ATMs, elevators
- For systems of this sort, we are interested in verifying properties (typically, about infinite behaviours) like the following:
  - Can the system ever get into a certain state?
  - If it ever gets into state A, can it eventually then go into state B?
  - From any state it gets into, will it always get back to a starting state?
- One technique that has been found useful for verifying properties like these is the following:
  - model the system as a *structure* of a certain logical language
  - formulate the property to be checked as a *sentence* in the language
  - determine whether or not the sentence is *true* in the structure.
- This is called model-checking.

Overview of Model Checking



Model Checking in Industry

- Microprocessor design: Several major microprocessor manufacturers use model checking methods as a part of their design process.
- Design of Data-communications Protocol Software: Model checkers have been used as rapid prototyping systems for new data-communications protocols under standardisation.
- Mission Critical Software: NASA space program is model checking code used by the space program.
- Operating Systems: Microsoft is using model checking to verify the correct use of locking primitives in Windows device drivers.

Modeling

As a language describing system models we can for example use:

- Petri nets,
- labelled transition systems (LTSs) and process algebras,
- Java programs,
- UML (unified modelling language) state machines,
- Promela language (input language of the Spin model checker), and
- VHDL,Verilog, or SMV languages (mostly for HW design).

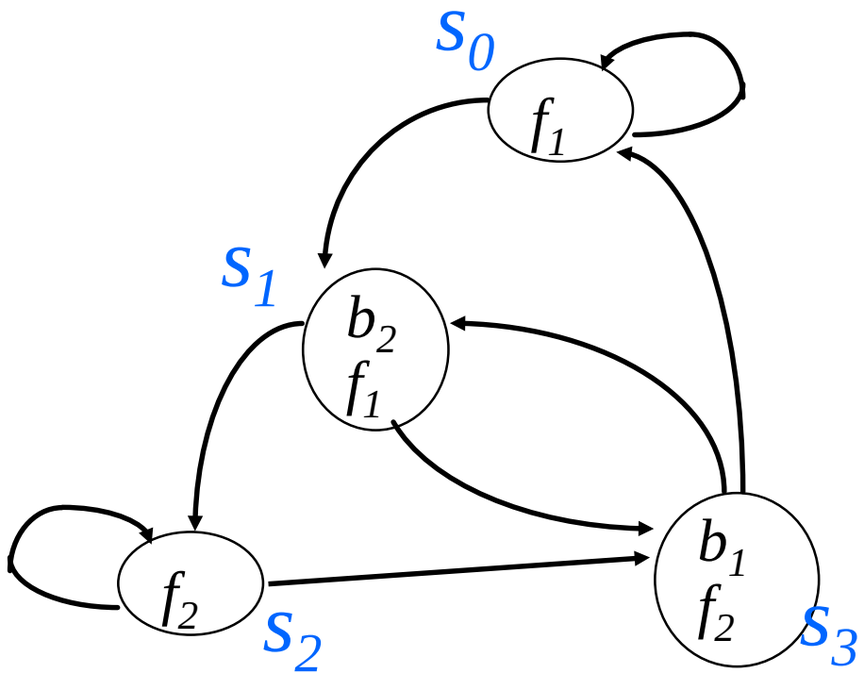
Some Model Checking Approaches

- Explicit State Model Checking: Tools include Spin, Murφ Java Pathfinder Maria, PROD, CPN Tools, CADP, etc.
- BDD based Symbolic Model Checking: Tools include NuSMV 2, VIS, Cadence SMV, etc.
- Bounded Model Checking: Tools include BMC, CMBC, NuSMV 2, VIS, Cadence SMV, etc.

Models: Kripke Structures

- *State Transition Systems*
  - *K = <V, S, s_0, I , R>*
  - *V* is a (finite) set of atomic propositions, e.g., *{p, q, r, f, b}*
    - "Call button is on"
    - "There are no requested jobs for the printer"
    - "Conveyer belt is stopped"

  Should not involve time!



  - *S* is a (finite) set of states
  - $s_0 \in S$ is a start state
  - $I: S \rightarrow 2^V$ is a labeling function that maps each state to the set of propositional variables that hold in it

  Alternatively: a set of interpretations specifying which propositions are true in each state

  - $R \subseteq S \times S$ is a transition relation over *S*

Paths

- A path over $\langle S, s_0, \mathrm{R} \rangle$ is a non-empty sequence h of states such that if $h = \ldots s \cdot s' \ldots$, then $(s, s') \in \mathrm{R}$.

  For example, consider the system $\langle S, s_0, \mathrm{R} \rangle$
  where $S = \{s_0, s_1\}$ and
  $\mathrm{R} = \{ (s_0, s_0), (s_0, s_1), (s_1, s_0) \}$

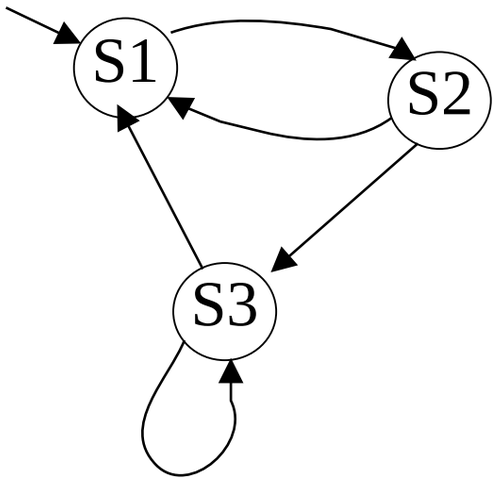
  Then here are some paths over this system:

  - s_1
  - $s_1 \cdot s_0$
  - $s_0 \cdot s_1 \cdot s_0 \cdot s_0$
  - $s_0 \cdot s_0 \cdot s_0 \cdot s_0 \cdot s_0 \cdot s_0 \cdot \ldots$
  - $s_0 \cdot s_1 \cdot s_0 \cdot s_0 \cdot s_1 \cdot s_0 \cdot s_0 \cdot s_0 \cdot s_1 \cdot s_0 \cdot s_0 \cdot s_0 \cdot s_0 \cdot s_1 \cdot \ldots$

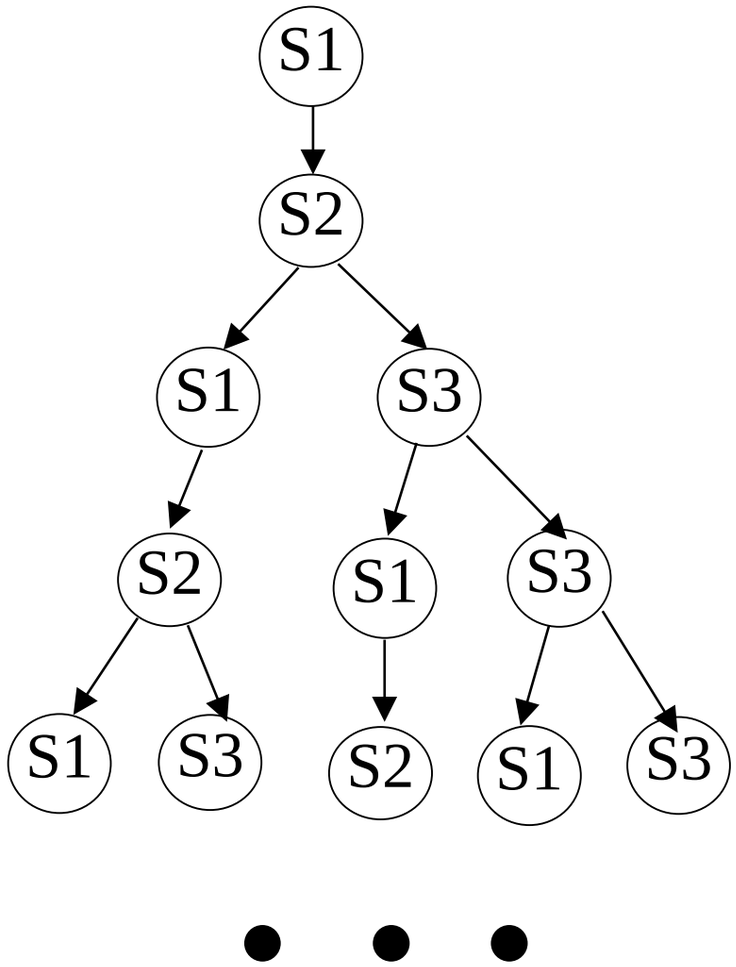
  The last two are *infinite* paths

Computation Tree Logic (CTL)

- CTL: Branching-time propositional temporal logic
- Model - a tree of computation paths
- Example:



State-transition graph
(called Kripke structure)



Tree of computation

CTL: Computation Tree Logic

- Def. *CTL* is a branching-time temporal logic. It allows explicit quantification over possible futures. CTL is a language without quantifiers that includes the usual boolean connectives as well as the following temporal connectives:

  EX p : p holds in some next states
  (translated as $\exists s\ [Next(s,start) \wedge P(s)]$)

  EF p: along some path, p is true in a future state

  E[p U q] : along some path, p holds until q holds

  EG p : along some path, p holds in every state

  AX p: p holds in all next states

  AF p: along all paths, p is true in a future state

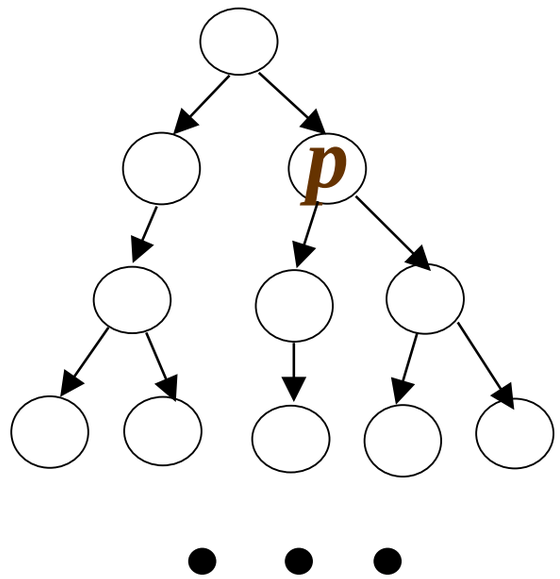
  A[p U q]: along all paths, p holds until q holds

  AG p: along all paths, p holds in every state

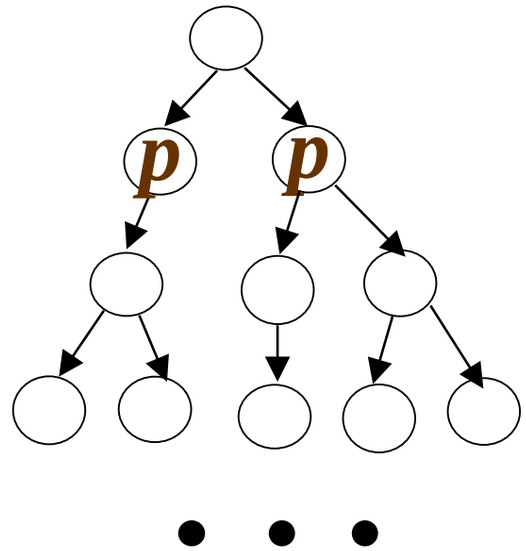
  A state is mentioned explicitly or else the property is assumed to be about the initial state

- *More compact representation than first-order logic but less expressive.*
- *Meant to reason only about infinite behaviours.*

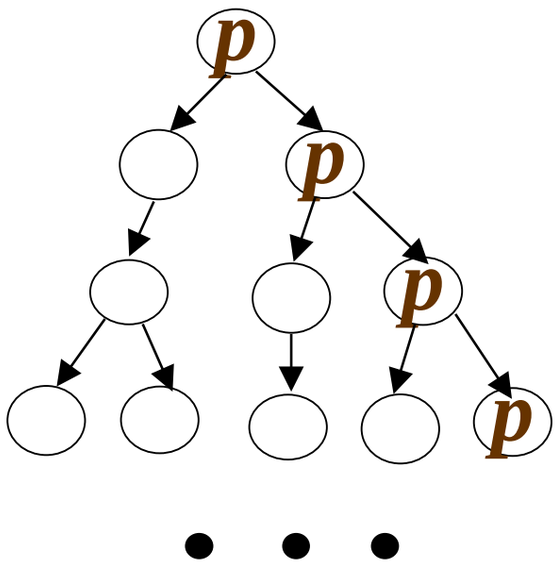
Examples



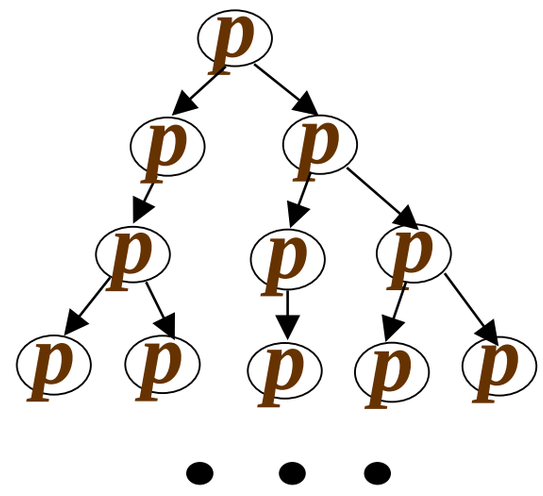
EX p (exists next)



AX p (all next)

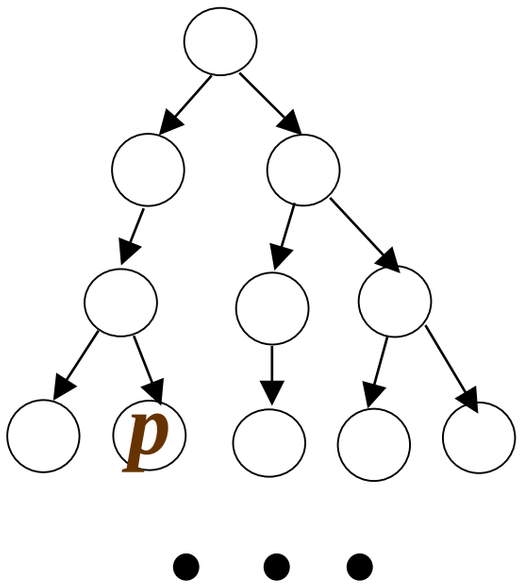


EG p (exists global)

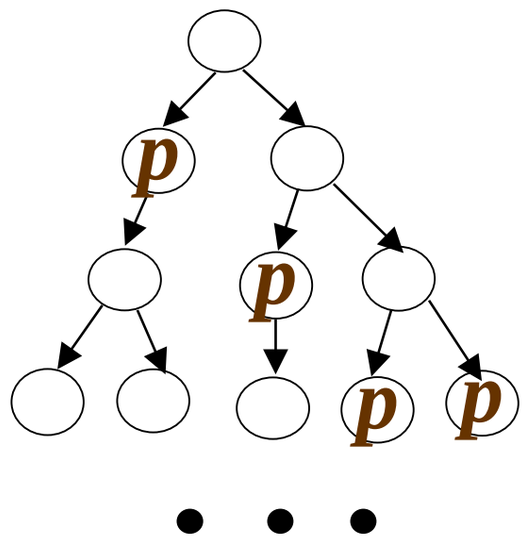


AG p (all global)

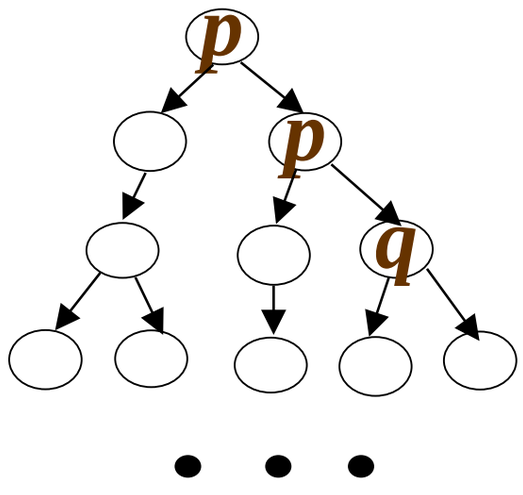
Examples (Cont'd)



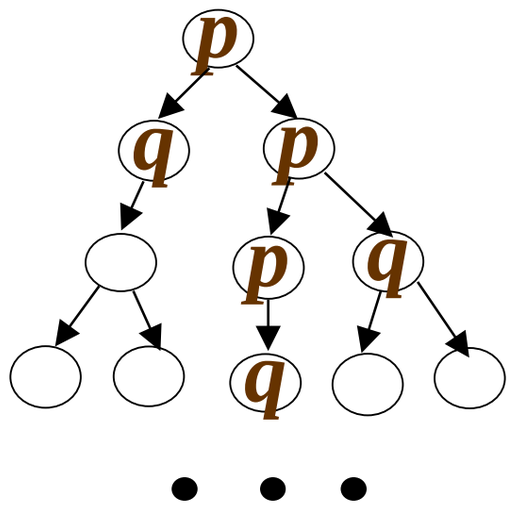
EF p (exists future)



AF p (all future)



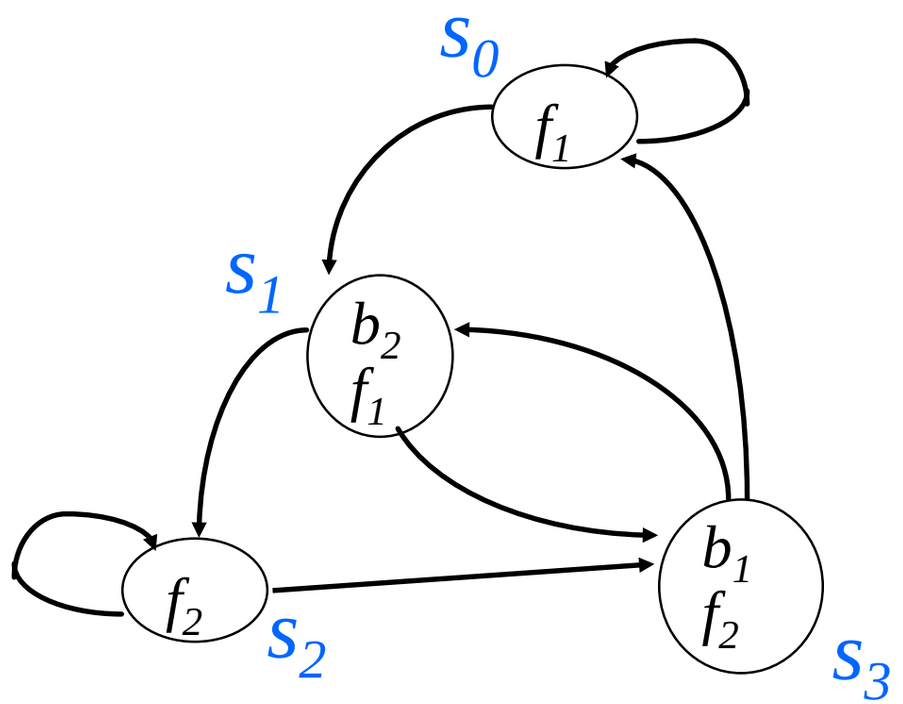
E[pUq] (exists until)



A[pUq] (all until)

CTL Examples

- Which of the properties hold?
  - (AX f_1)(s_0)
  - (EG f_2)(s_2)
  - A [b_1 U f_1] (s_3)
  - AG ($b_1 \Rightarrow$ AF f_1)
  - AG ($f_1 \vee f_2$)



Some More Statements To Express

- When a request occurs, it will eventually be acknowledged
  - AG (request ⇒ 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[¬*q* U (*q* ∧ A[¬*p* U *s*])]

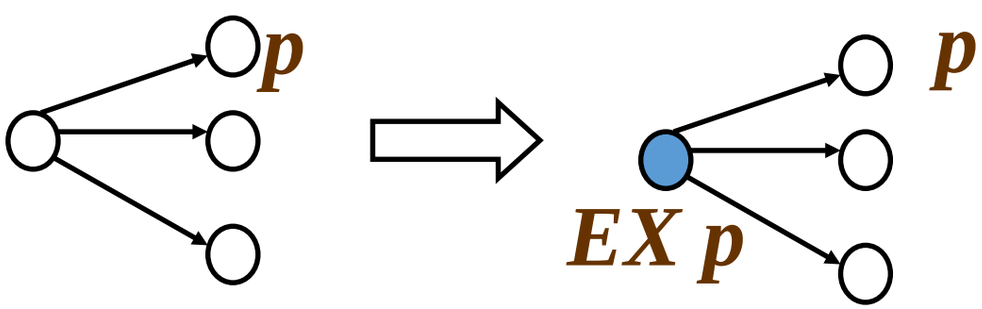
CTL Model-Checking

- Receive:
  - Kripke structure *K*
  - Temporal logic formula *f*
- Assumptions:
  - Finite number of processes
    - Each having a finite number of finite-valued variables
  - Finite length of a CTL formula
- Algorithm:
  - Label states of *K* with subformulas of *f* that are satisfied there and working outwards towards *f*.
  - Output states labeled with *f*

  Example: EX EG ($p \Rightarrow$ E[p U q])

CTL Model-Checking (Cont'd)

- EX *p*
  - Label any state with EX *p* if any of its successors are labeled with *p*



- E [*p* U *q*]
  - If any state s is labeled with *q*, label it with E[*p* U *q*]
  - Repeat:

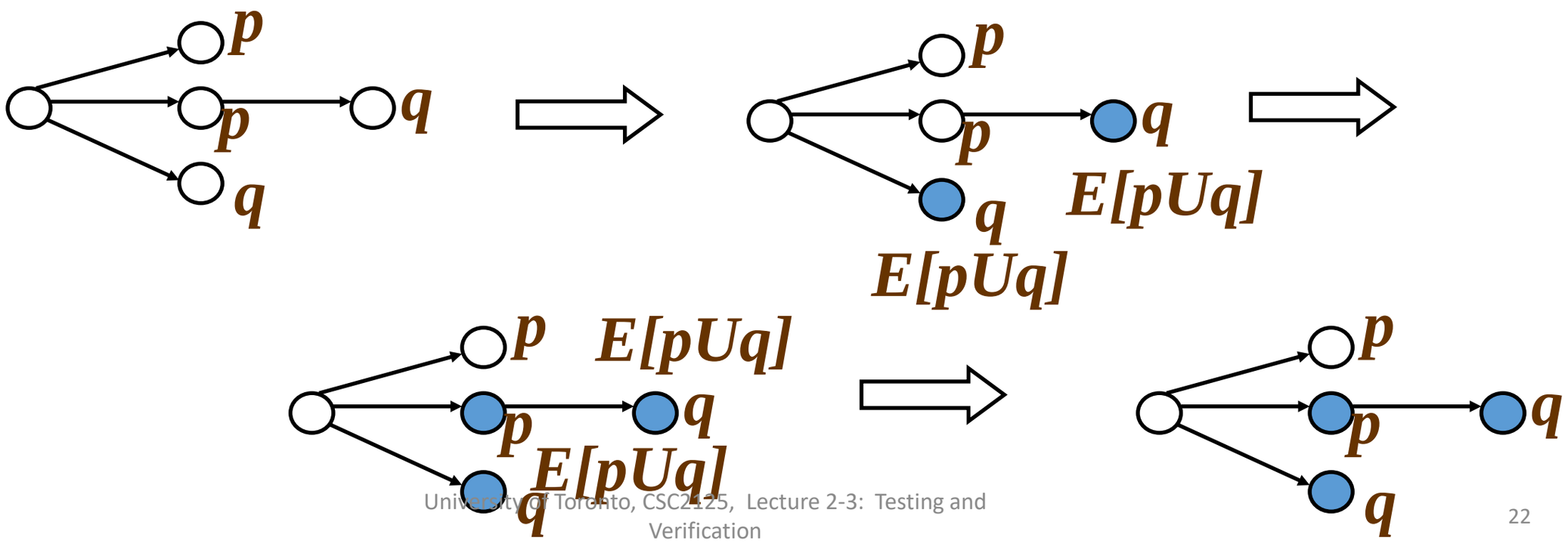
    label any state with E[*p* U *q*]

    if it is labeled with *p* and at least one of its successors is labeled with E[*p* U *q*]

  until there is no change

Fragment of Kripke structure.

Dark dots are states labelled with appropriate subformula

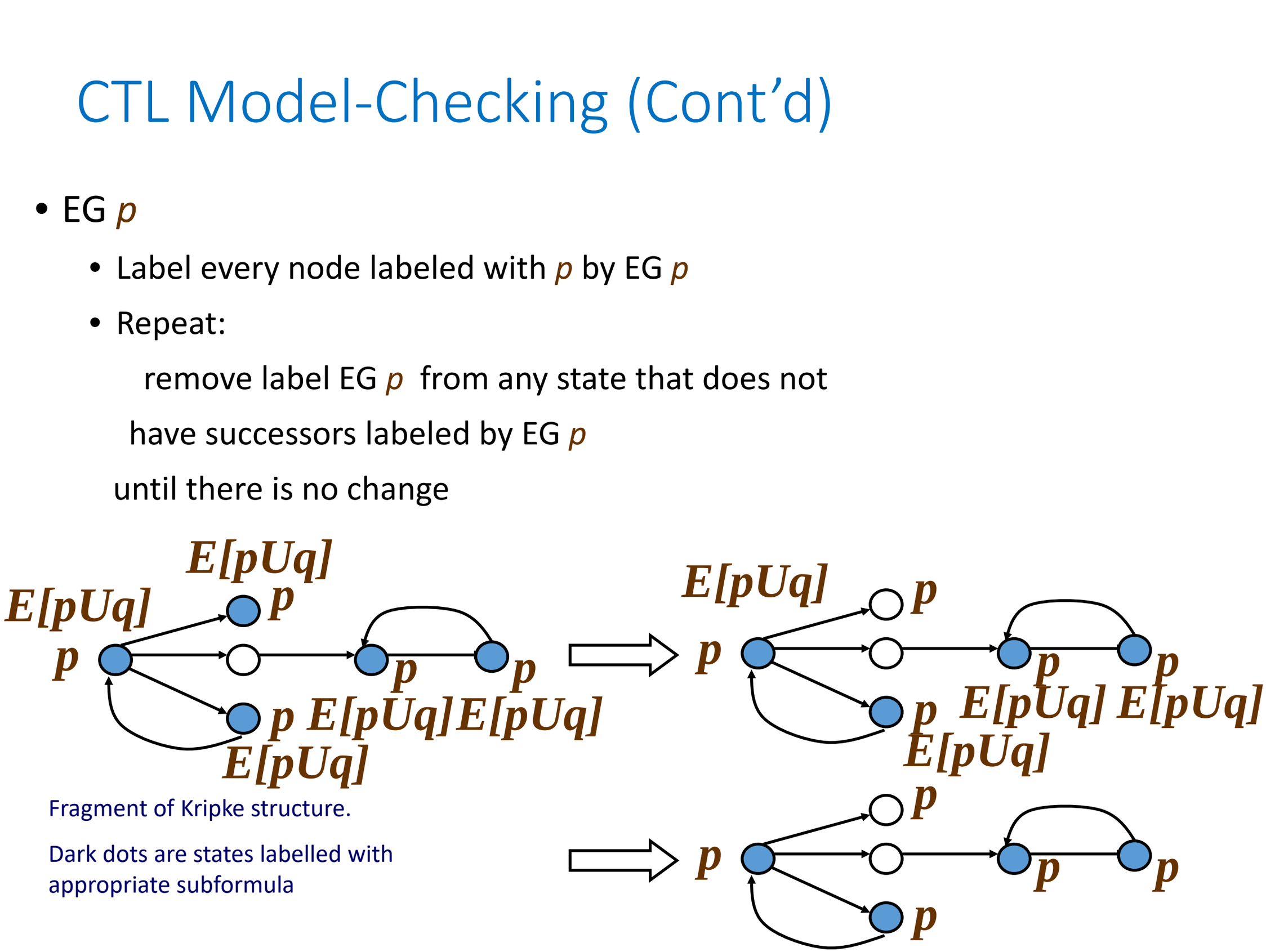


CTL Model-Checking (Cont'd)

- EG *p*
  - Label every node labeled with *p* by EG *p*
  - Repeat:

    remove label EG *p* from any state that does not have successors labeled by EG *p*

    until there is no change

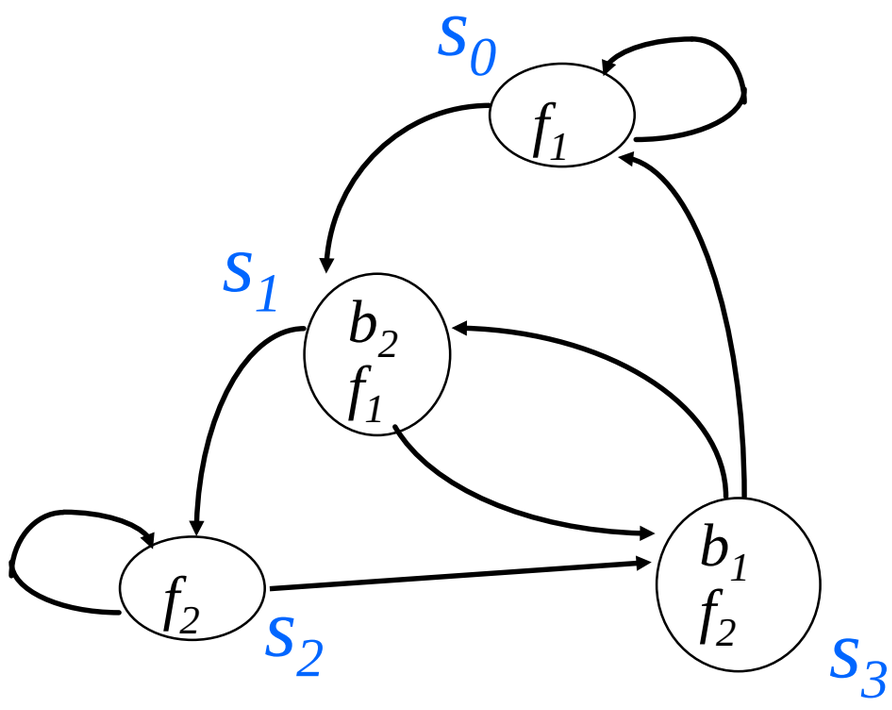


Fragment of Kripke structure.

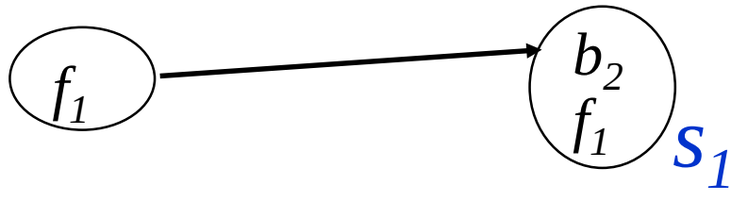
Dark dots are states labelled with appropriate subformula

Counterexamples

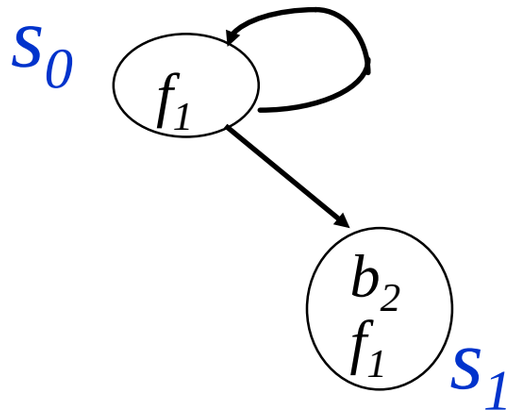
- Explain why the property fails to hold
- to disprove that ϕ holds on all elements of S, produce a single element $s \in S$ s.t. $\neg\phi$ holds on s.
  - counterexamples restricted to universally-quantified formulas
  - counterexamples are paths (trees) from initial state illustrating the failure of property



- AG $\neg b_2$



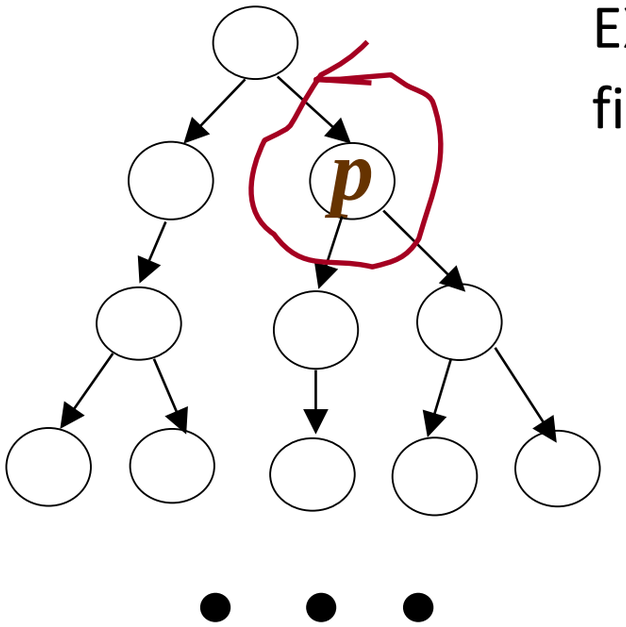
- AF b_2 $\vee$ AX $\neg b_2$



Generating Counterexamples

Negate the prop. and express using EX, EU, EG

- e.g., AG ($p \Rightarrow$ AF q) becomes EF($p \wedge$ EG $\neg q$)

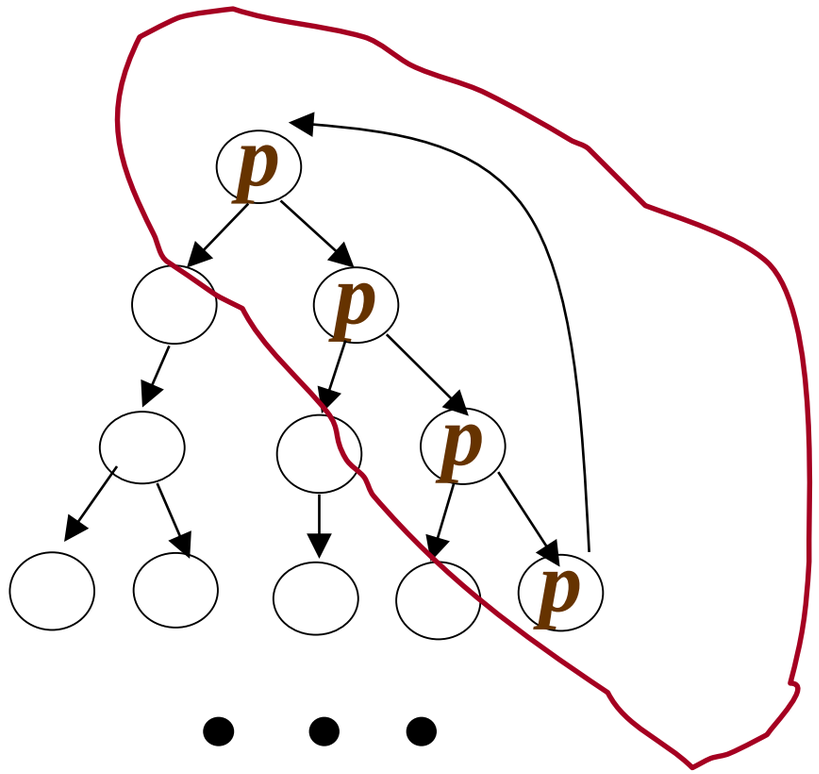


EX *p* :

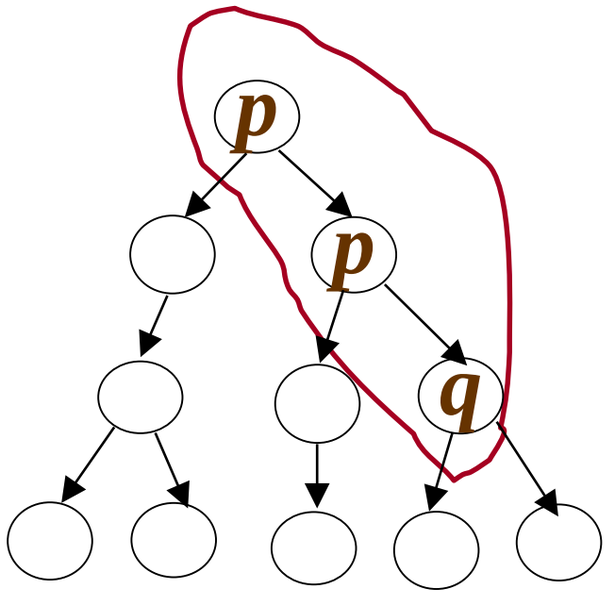
find a successor state labeled with *p*

EG *p:*

follow successors labeled with EG *p* until a loop is found



Generating Counterexamples (Cont'd)



E[*p* U *q*]:
remove all states not labeled with
p or *q*, then look for path to *q*

● ● ●

- This procedure works only for universal properties
  - AX *p*
  - AG (*p* ⇒ AF *q*)
  - etc.

State Explosion

- How fast do Kripke structures grow?
  - Composing linear number of structures yields exponential growth!
  - Models of size 2^{100} are very easy to obtain
- 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
- *Composition*
  - Break the verification problem down into several simpler verification problems

Symbolic Model Checking

- Why?
  - Saves us from constructing a model state space explicitly. Effective “cure” for state space explosion.
- How?
  - Sets of states and the transition relation are represented by formulas. Set operations are defined in terms of formula manipulations

Representing Models Symbolically

- A system state represents an interpretation (truth assignment) for a set of propositional variables *V*

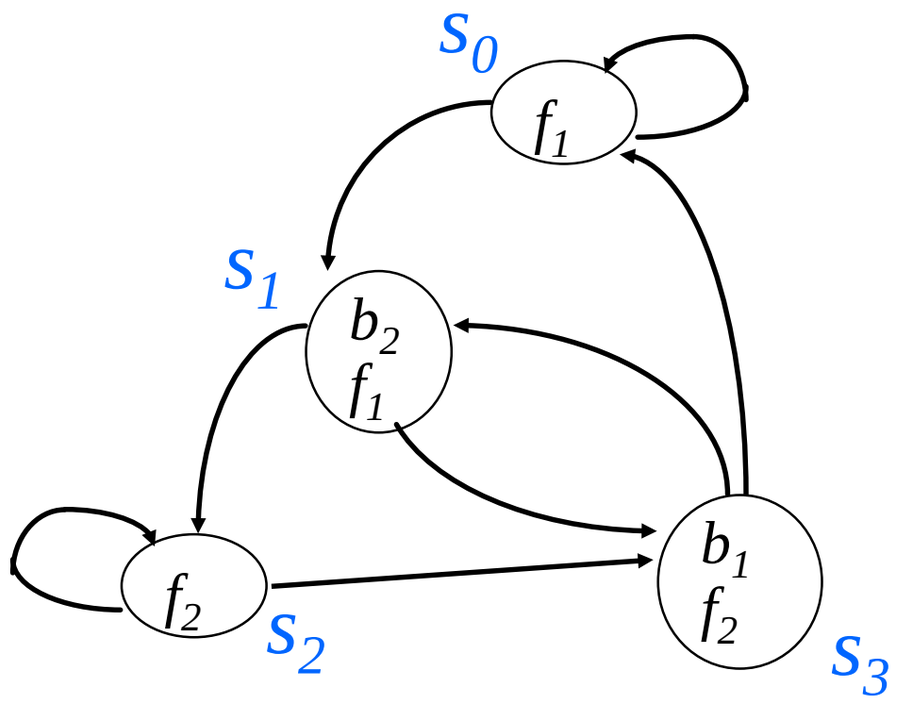
Formulas represent sets of states that satisfy it

False = ∅, *True* = *S*

f_1 – set of states in which f_1 is true – $\{s_0, s_1\}$

f_2 – set of states in which f_2 is true – $\{s_2, s_3\}$

$f_1 \vee f_2 = \{s_0, s_1, s_2, s_3\} = S$



State transitions are described by relations over two sets of variables: *V* (source state) and *V'* (destination state)

Transition (s_2, s_3) is $\neg b_1 \wedge \neg b_2 \wedge \neg f_1 \wedge f_2 \wedge b_1' \wedge \neg b_2' \wedge \neg f_1' \wedge f_2'$

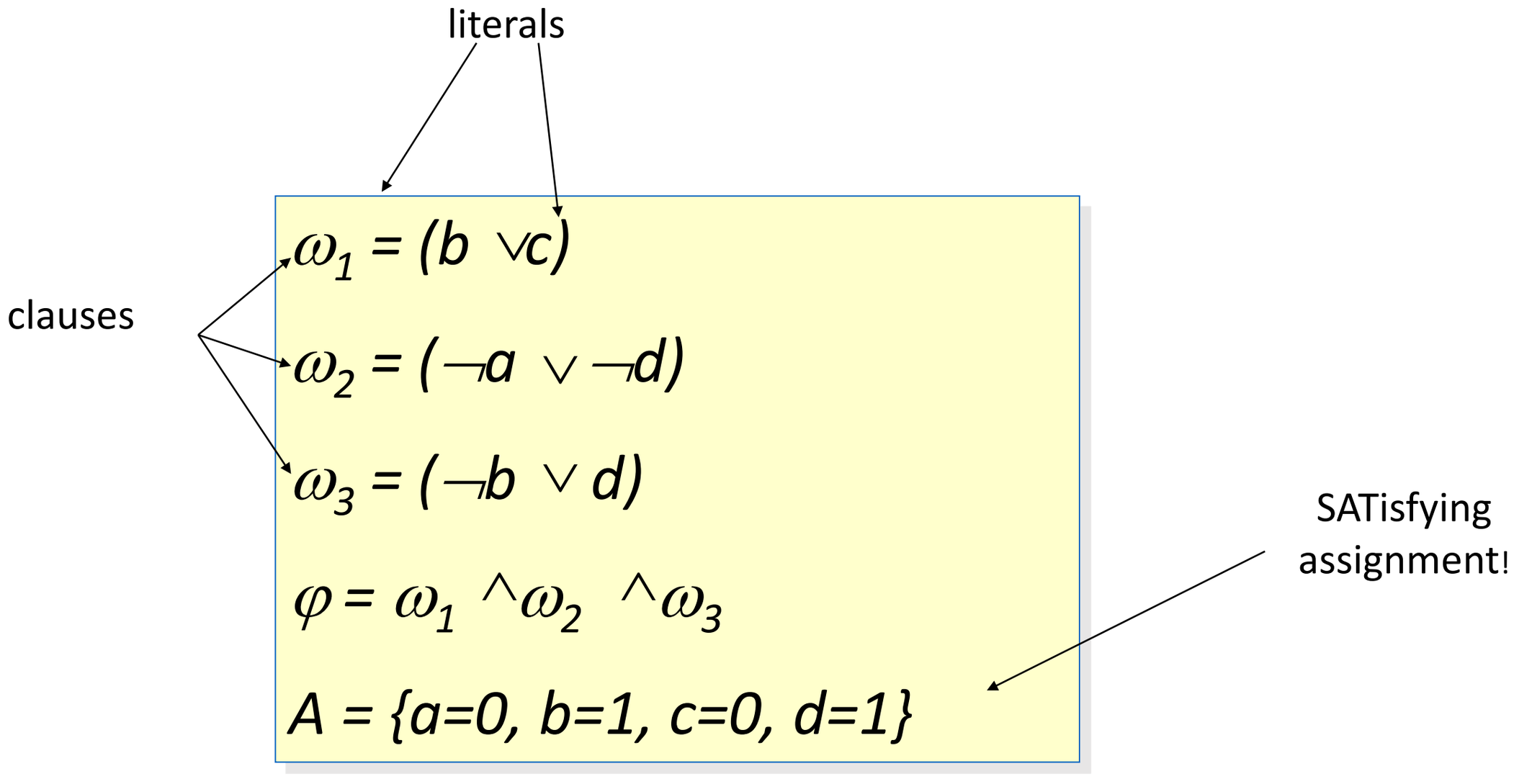
Relation *R* is described by disjunction of formulas for individual transitions

A slight aside – SAT solving

- The propositional satisfiability problem (SAT) is one of the main instances of NP-complete problems.
- Thus no polynomial algorithms for SAT are known.
- However, there are highly efficient SAT solvers available such as zChaff and MiniSAT which are able solve many bounded model checking problems efficiently.

What is SAT?

Given a propositional formula in CNF, find if there exists an assignment to Boolean variables that makes the formula true:



DPLL: Historical Perspective

- 1962: the original algorithm known as DP (Davis-Putnam)
  $\Rightarrow$ "simple" procedure for automated theorem proving

- Davis and Putnam hired two programmers, George Logemann and David Loveland, to implement their ideas on the IBM 704.

- Not all of their ideas worked out as planned $\Rightarrow$ refined algorithm to what is known today as DPLL

DPLL Insight

- There are two distinct ways to approach the boolean satisfiability problem:
- Search
  - Find satisfying assignment in by searching through all possible assignments ⇒ most basic incarnation: truth table!
- Deduction
  - Deduce new facts from set of known facts ⇒ application of proof rules, semantic argument method
- DPLL combines search and deduction in a very effective way!

Deduction in DPLL

- Deductive principle underlying DPLL is propositional resolution
- Resolution can only be applied to formulas in CNF
- SAT solvers convert formulas to CNF to be able to perform resolution

Propositional Resolution

- Consider two clauses in CNF:

$$C_1: \quad (l_1 \vee \ldots p \ldots \vee l_k) \qquad C_2: \quad (l'_1 \vee \ldots \neg p \ldots \vee l'_n)$$

- From these, we can deduce a new clause C_3, called resolvent:

$$C_3: \quad (l_1 \vee \ldots \vee l_k \vee l'_1 \vee \ldots\ldots \vee l'_n)$$

- Correctness:
  - Suppose p is assigned $\top$: Since C_2 must be satisfied and since $\neg p$ is $\bot$, $(l'_1 \vee \ldots\ldots \vee l'_n)$ must be true.
  - Suppose p is assigned $\bot$: Since C_1 must be satisfied and since p is $\bot$, $(l_1 \vee \ldots\ldots \vee l_k)$ must be true.
  - Thus, C_3 must be true.

Unit Resolution

- DPLL uses a restricted form of resolution, known as unit resolution.
- Unit resolution is propositional resolution, but one of the clauses must be a unit clause (i.e., contains only one literal)
- $C_1 : p \qquad C_2 : \quad (l_1 \vee \ldots \neg p \ldots \vee l_n)$
- Resolvent: $(l_1 \vee \ldots \vee l_n)$
- Performing unit resolution on C_1 and C_2 is same as replacing p with true in the original clauses.
- In DPLL, all possible applications of unit resolution called Boolean Constraint Propagation (BCP).

Boolean Constraint Propagation (BCP) Example

- Apply BCP to CNF formula:

$$(p) \wedge (\neg p \vee q) \wedge (r \vee \neg q \vee s)$$

- Resolvent of first and second clause: q
- New formula: $q \wedge (r \vee \neg q \vee s)$
- Apply unit resolution again: $(r \vee s)$
- No more unit resolution possible, so this is the result of BCP.

Basic DPLL

```
bool DPLL(φ)
{
  1. φ' = BCP(φ)
  2. if(φ' = ⊤) then return SAT;
  3. else if(φ' = ⊥) then return UNSAT;
  4. p = choose_var(φ');
  5. if(DPLL(φ'[p ↦ ⊤])) then return SAT;
  6. else return (DPLL(φ'[p ↦ ⊥]));
}
```

- Recursive procedure; input is formula in CNF
- Formula is $\top$ if no more clauses left
- Formula becomes $\bot$ if we derive $\bot$ due to unit resolution

An Optimization: Pure Literal Propagation

- If variable p occurs only positively in the formula (i.e., no $\neg p$), p must be set to $\top$
- Similarly, if p occurs only negatively (i.e., only appears as $\neg p$), p must be set to $\bot$
- This is known as Pure Literal Propagation (PLP).

DPLL with Pure Literal Propagation

```
bool DPLL(φ)
{
  1. φ' = BCP(φ)
  2. φ'' = PLP(φ')
  3. if(φ'' = ⊤) then return SAT;
  4. else if(φ'' = ⊥) then return UNSAT;
  5. p = choose_var(φ'');
  6. if(DPLL(φ''[p ↦ ⊤])) then return SAT;
  7. else return (DPLL(φ''[p ↦ ⊥]));
}
```

Example

$$F: (\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$$

- No BCP possible because no unit clause
- No PLP possible because there are no pure literals
- Choose variable q to branch on:

$$F[q \mapsto \top]: (r) \wedge (\neg r) \wedge (p \vee \neg r)$$

- Unit resolution using (r) and $(\neg r)$ deduces $\bot \Rightarrow$ backtrack

Example Cont.

$$F: (\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$$

- Now, try $q = \bot$

$$F[q \mapsto \bot]: (\neg p \vee r)$$

- By PLP, set p to $\bot$ and r to $\top$
- $F[q \mapsto \bot, p \mapsto \bot, r \mapsto \top] : \top$
- Thus, F is satisfiable and the assignment $[q \mapsto \bot, p \mapsto \bot, r \mapsto \top]$ is a model (i.e., a satisfying interpretation) of F.

SAT-Based Model-Checking

- Expand transition relation a fixed number of steps (e.g., loop unrolling), resulting in a formula
- For this unrolling, check whether the property holds
- Continue increasing the unrolling until error is found, resources are exhausted, or diameter of the problem is reached
- Based on very fast SAT solvers (e.g., ZChaff)

Bounded Model Checking

An approach to symbolic model-checking that uses SAT solvers. It is called *Bounded Model Checking*.

Applications:

- A.I. Planning problems: *can we reach a desired state in k steps?*
- Verification of *safety* properties: *can we find a bad state in k steps?*
- Verification: *can we find a counterexample in k steps?*

Given: transition system *M*, temporal logic formula *f*, user-supplied time bound *k*

Construct: propositional formula $\Omega(k)$ that is *satisfiable* iff *f* is valid along a path of length *k*

Path of length *k*: $$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$$

starting from the initial state, go *k* steps forward

Say *f* = **EF** *p* (*p* is reachable) and *k* = 2, then

$$\Omega(2) = I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2) \wedge (p_0 \vee p_1 \vee p_2)$$

BMC idea: Checking Invariants

AG *p* means *p* must hold in every state along any path of length *k*

We take

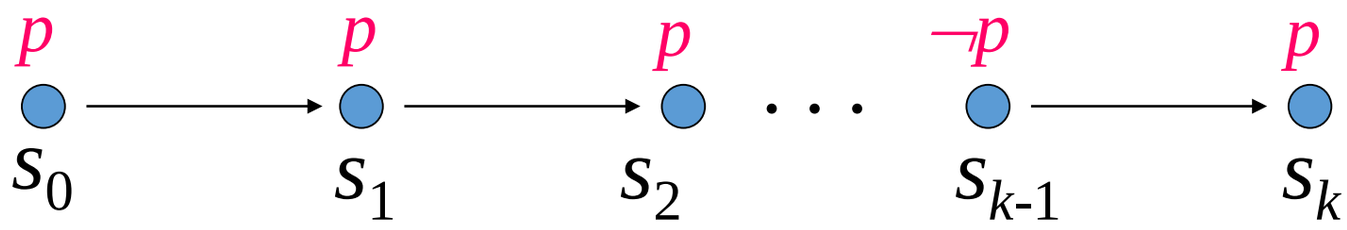
$$\neg\Omega(k) = (I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})) \rightarrow \bigwedge_{i=0}^{k} p_i$$

So

$$\Omega(k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p_i$$

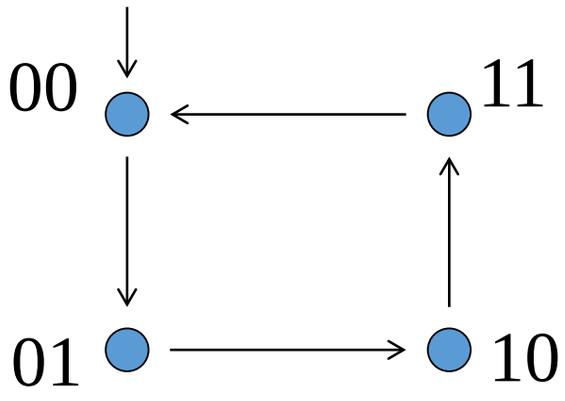
p is preserved up to *k-th* transition iff $\Omega(k)$ is unsatisfiable:

$$\Omega(k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p$$



If satisfiable, satisfying assignment gives counterexample to the safety property.

Example: A two-bit counter



Initial state: $I : \neg l \wedge \neg r$

Transition: $R: \begin{pmatrix} l' = (l \neq r) \wedge \\ r' = \neg r \end{pmatrix}$

Safety property: **AG** $(\neg l \vee \neg r)$

$$\Omega(2) : (\neg l_0 \wedge \neg r_0) \wedge \begin{pmatrix} l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0 \wedge \\ l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1 \end{pmatrix} \wedge \begin{pmatrix} (l_0 \wedge r_0) \vee \\ (l_1 \wedge r_1) \vee \\ (l_2 \wedge r_2) \end{pmatrix}$$

$\Omega(2)$ is unsatisfiable.

$\Omega(3)$ is satisfiable.

Unbounded Model Checking

- A variety of methods to exploit SAT and BMC for unbounded model checking:
  - Completeness Threshold
  - *k* - induction
  - Abstraction (refutation proofs useful here)
  - Exact and over-approximate image computations (refutation proofs useful here)
  - Use of Craig interpolation

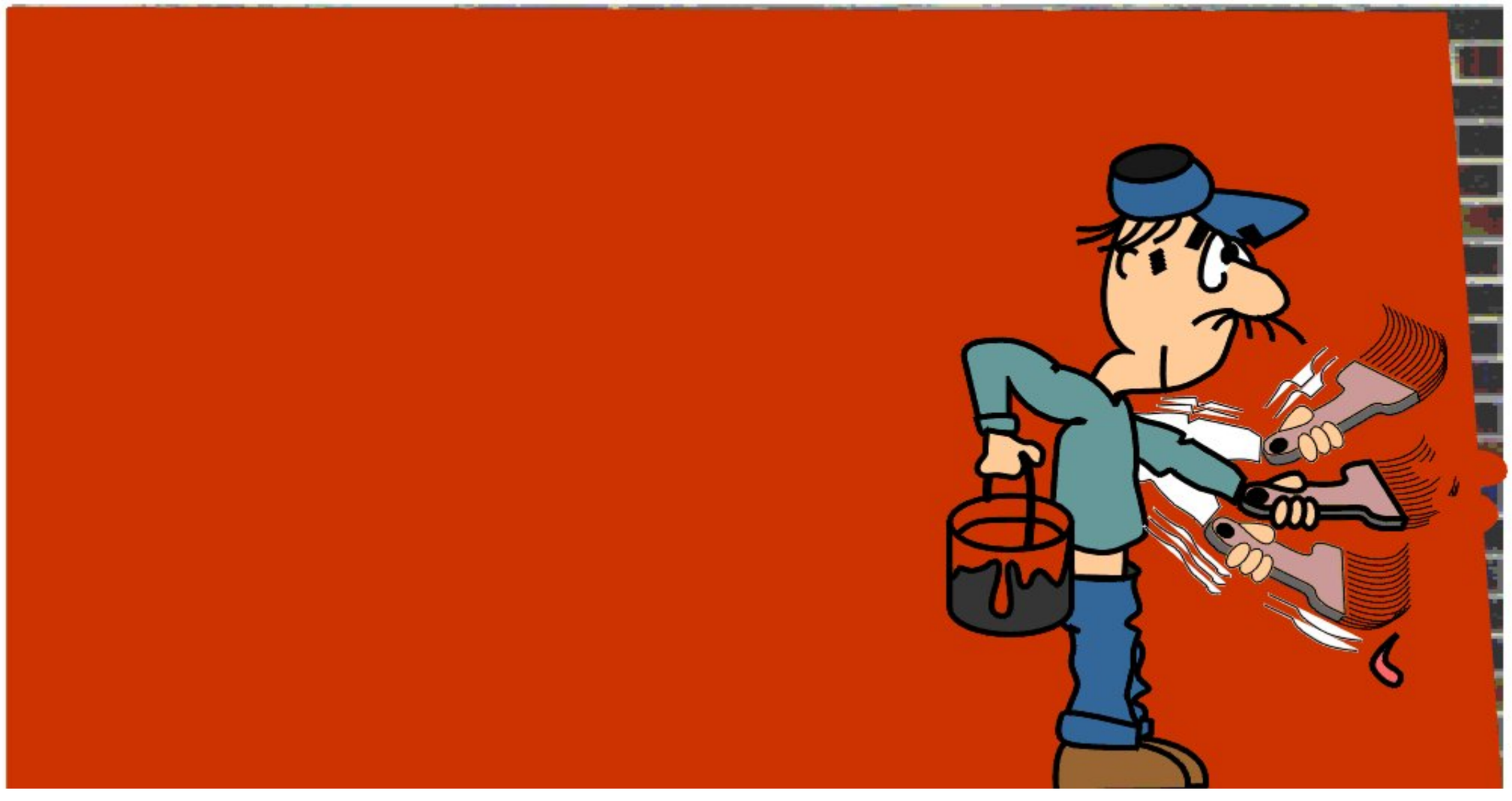
Pros and Cons of Model-Checking

- Often cannot express full requirements
  - Instead check several smaller properties
- Largely automatic and fast
- Few systems can be checked directly
  - Must generally abstract
- Work better for certain types of problems
  - Very useful for control-centered concurrent systems, such as avionics software, hardware, communication protocols
  - Not very good at data-centered systems such as user interfaces and databases
- Better use for debugging rather than assurance
- Testing vs model-checking
  - Usually, find more problems by exploring all behaviours of a downscaled system than by testing some behaviours of the full system
  - Bounded exploration – all behaviors of a downscaled system up to a particular depth

(White-Box) Testing Primer

Code Coverage

Introduced by Miller and Maloney in 1963



Coverage Criteria

Basic Coverage

Advanced Coverage

- **Line coverage**
- **Statement**
- **Function/Method coverage**
- **Branch coverage**
- **Decision coverage**
- **Condition coverage**
- **Condition/decision coverage**
- **Modified condition/decision coverage**
- **Path coverage**
- **Loop coverage**
- **Mutation adequacy**
- **…**

Line Coverage

- Percentage of source code lines executed by test cases.
  - For developer easiest to work with
  - Precise percentage depends on layout?
    - var x = 10; if (z++ < x) y = x+z;
    - Requires mapping back from binary?
- In practice, coverage not based on lines, but on *control flow graph*

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(5);
}
// 100% statement coverage
```

- Coverage:

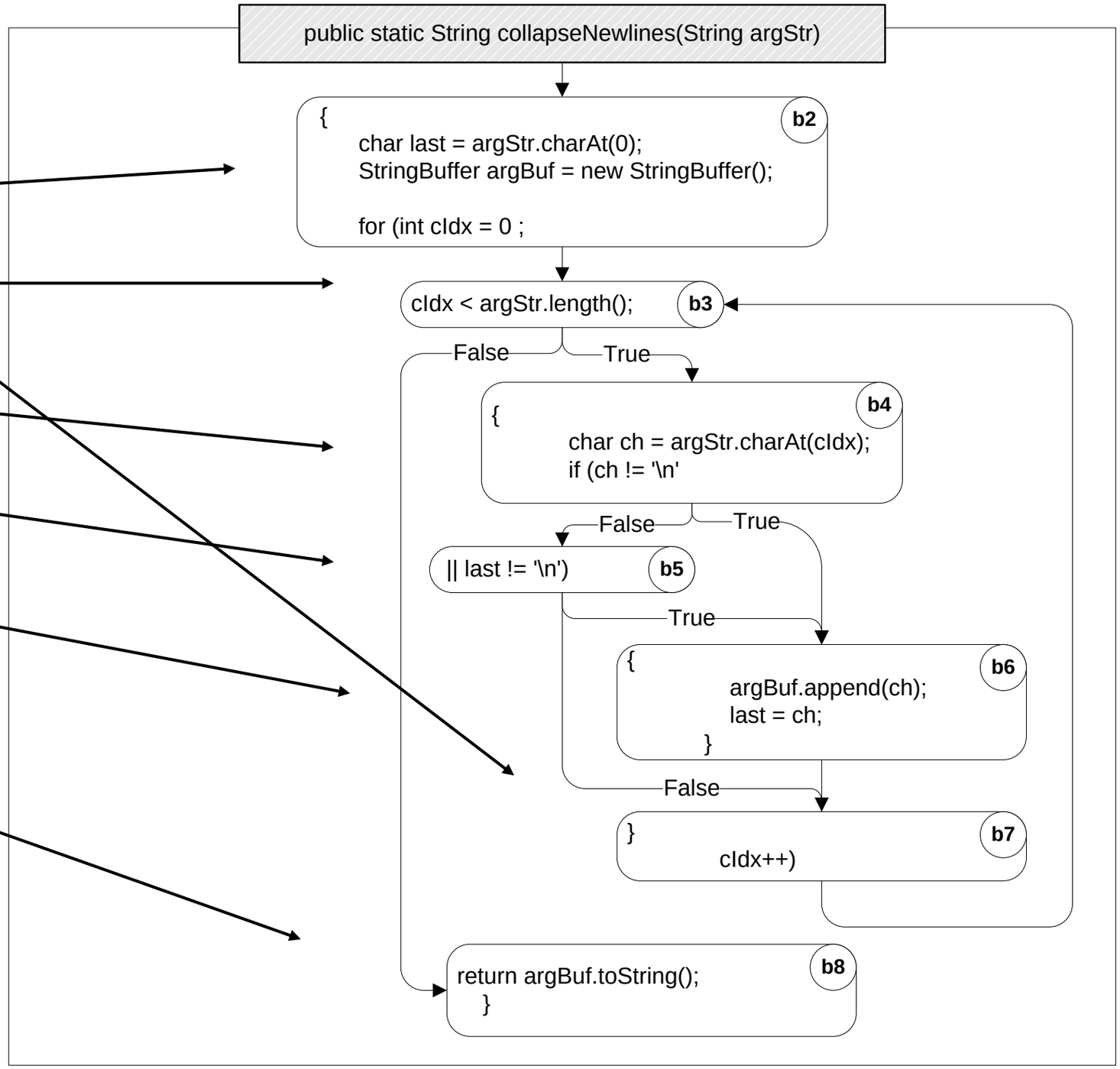
$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

Deriving a Control Flow Graph

```
public static String collapseNewlines(String argStr)
  {
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0; cIdx < argStr.length(); cIdx++)
    {
      char ch = argStr.charAt(cIdx);
      if (ch != '\n' || last != '\n')
      {
        argBuf.append(ch);
        last = ch;
      }
    }

    return argBuf.toString();
  }
```

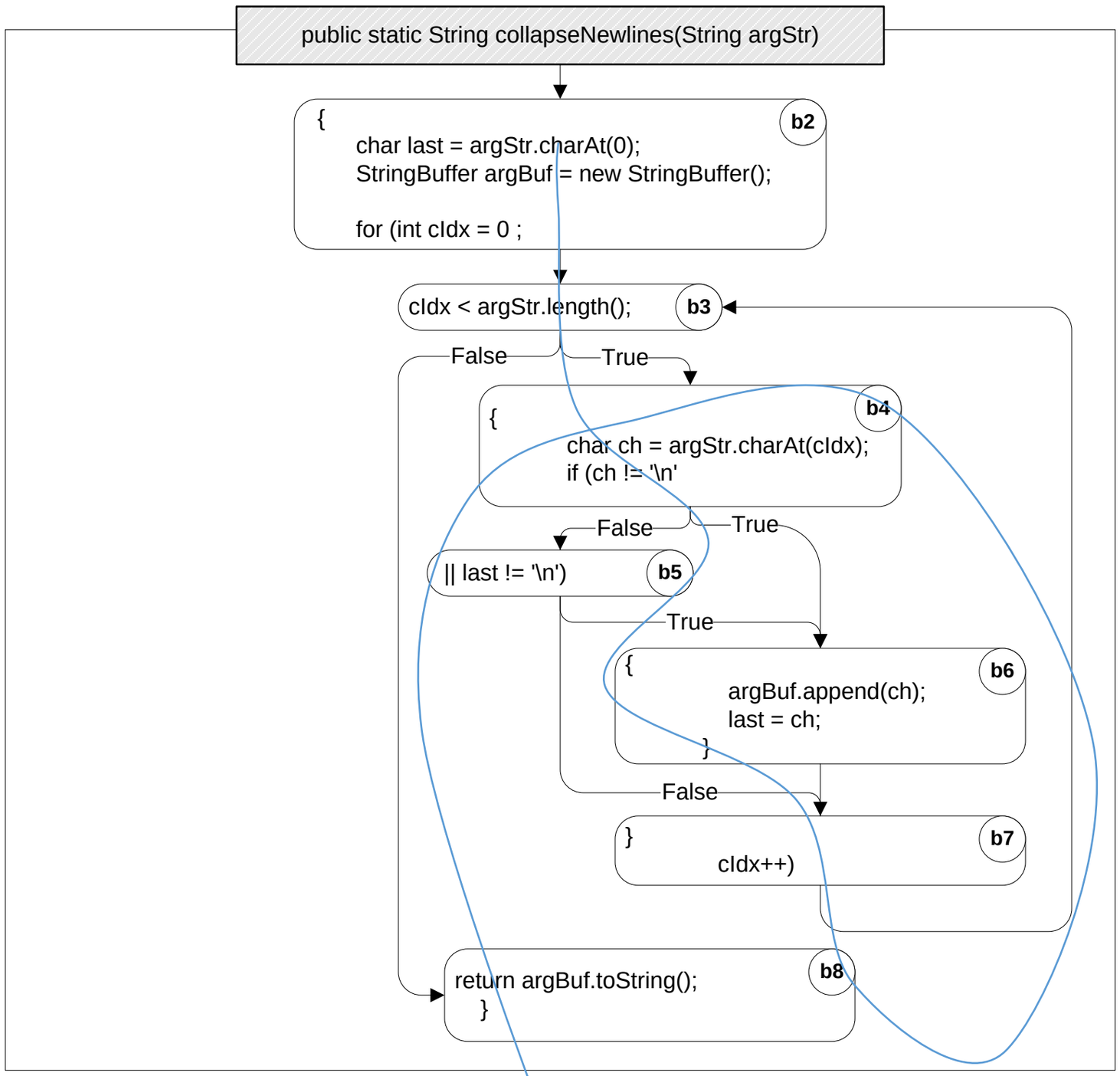


Splitting multiple conditions depends on goal of analysis

Control Flow Based Adequacy Criteria

- Every block / Statement?

One test case: b2,3,4,5,6,7,**3,**8
Input: “a”



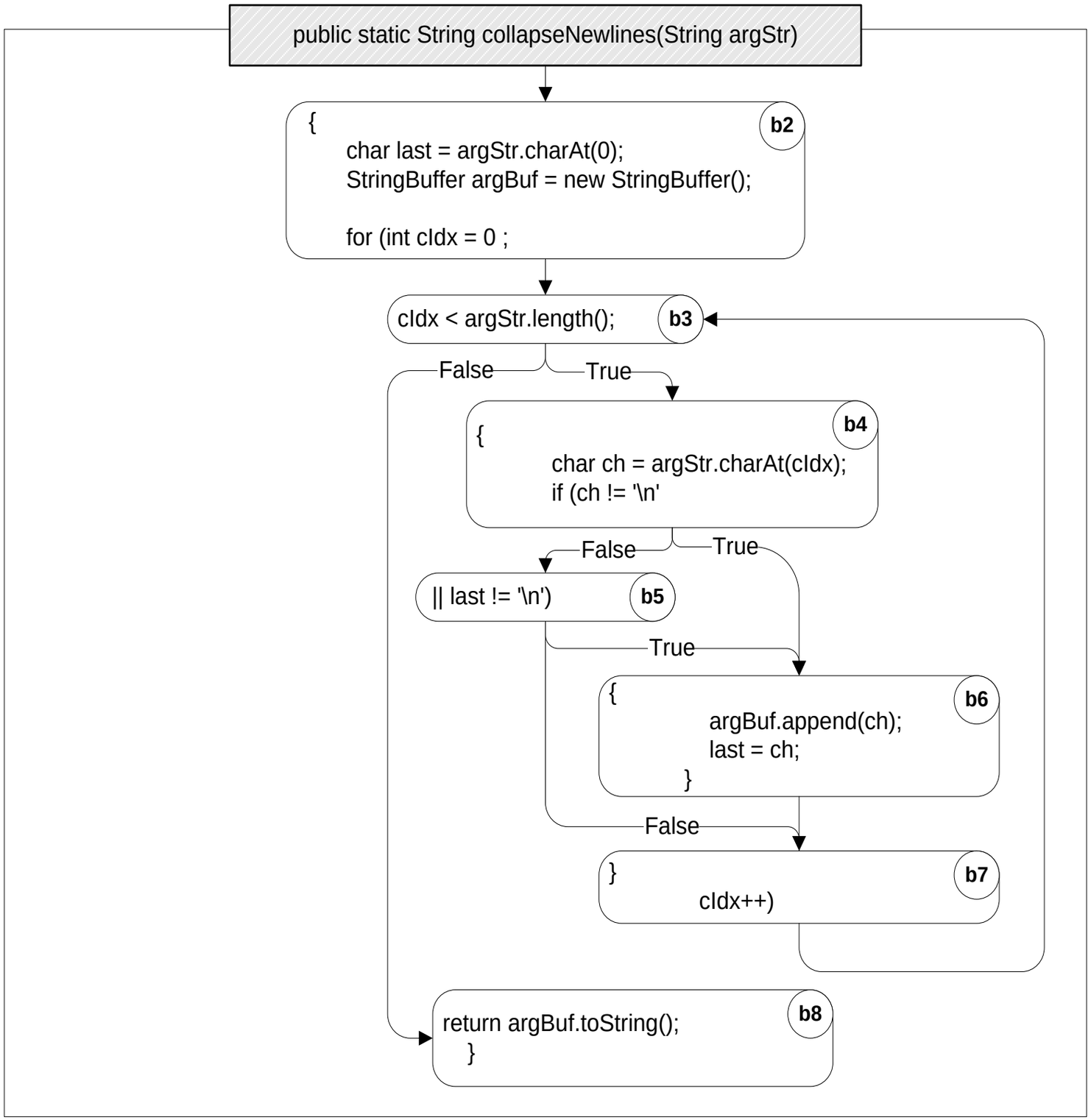
Branch Coverage

- Every path going out of node executed at least once
  - Decision-, all-edges-, coverage
  - Coverage: percentage of edges hit.

- *Each predicate must be both true and false*

Branch Coverage

- One longer input:
  - “a\n\n”
- Alternatively:
  - Block (“a”) and
  - “\n” and
  - “\n\n”



Condition Testing

- **Compound predicates:**
  - (((a || b) && c) || d) && e

- Should we test the effect of *individual* conditions on the outcome?

1. *Basic **condition***: each cond. true, false
2. *Branch and condition*: same, + branch
3. *Compound condition*: each combination, 2^N (costly)
4. *Modified Condition / Decision Coverage* (MC/DC)

MC/DC: Modified Condition + Decision Coverage

- Basic condition + decision coverage + ...
  - *each basic condition should **independently** affect outcome of each decision*
- Requires:
  - For each basic condition C, two test cases,
  - values of all *evaluated* conditions except C are the same
  - compound condition as a whole evaluates to *true* for one and *false* for the other
  - N + 1 cases, for N conditions.

Example: Basic Condition Coverage

```
foo (A, B, C) {
    if ( (A || B) && C ) {
       /* statements*/
    }
    else {
      /* statements*/
    }
}
```

In order to ensure **Condition coverage** criteria for this example, A, B and C should be evaluated at least one time "true" and one time "false" during tests:

```
T1: foo(true, true, true)
     // A = true, B = true, C = true
T2: foo(false, false, false)
     // A = false, B = false, C = false
```

Example: Decision Coverage

```
if ( (A || B) && C ) {
   /* instructions */
}
else {
/* instructions */
}
```

In order to ensure **Decision coverage** criteria, the condition ((A or B) and C) should also be evaluated at least one time to "true" and one time to "false”:

A = true, B = true, C = true ---> "true"

A = false, B = false, C = false ---> "false"

Example: MC/DC

```
if ( (A || B) && C ) {
    /* instructions */
}
else {
/* instructions */
}
```

In order to ensure **MC/DC** criteria, each *boolean variable* should be evaluated one time to "true" and one time to "false", and this with *affecting* the decision's outcome:

A = true / B = false / C = true   ---> "true”
A = *false* / B = false / C = true   ---> "false”
A = false / B = *true* / C = true   ---> "true"
A = false / B = true / C = *false* ---> "false”

```
(((a || b) && c) || d) && e
```

#tc	a	b	c	d	e	outcome
t1	*T*	F	T	F	T	T
t2						
t3						
t4						
t5						
t6						
t7						
t8						
t9						
t10						

```
(((a || b) && c) || d) && e
```

#tc	a	b	c	d	e	outcome	
t1	***T***	F	T	F	T	T	
t2	***F***	F	T	F	T	F	
t3	F	***T***	T	F	T	T	
t4	F	***F***	T	F	T	F	=t2
t5	T	F	***T***	F	T	T	=t1
t6	T	F	***F***	F	T	F	
t7	-	-	F	***T***	T	T	
t8	T	F	F	**F**	T	F	=t6
t9	-	-	-	T	**T**	T	=t7
t10	-	-	-	-	**F**	F	

```
(((a || b) && c) || d) && e
```

#tc	a	b	c	d	e	outcome
t1	***T***	F	T	F	T	T
t2	***F***	F	T	F	T	F
t3	F	***T***	T	F	T	T
t6	T	F	***F***	F	T	F
t7	-	-	F	***T***	T	T
t10	-	-	-	-	**F**	F

DO-178B/ED-12B Software Considerations in Airborne Systems and Equipment Certification

Failure Condition	Software Level	Coverage
Catastrophic	A	MC/DC
Hazardous / Severe	B	Decision Coverage
Major	C	Statement Coverage
Minor	D	
No Effect	E	

- The worldwide avionics software standard which all airborne software is required to comply.
- The world's strictest software standard
- Influences other domains including medical devices, transportation, and telecommunications.

Coverage Criteria

Basic Coverage

Advanced Coverage

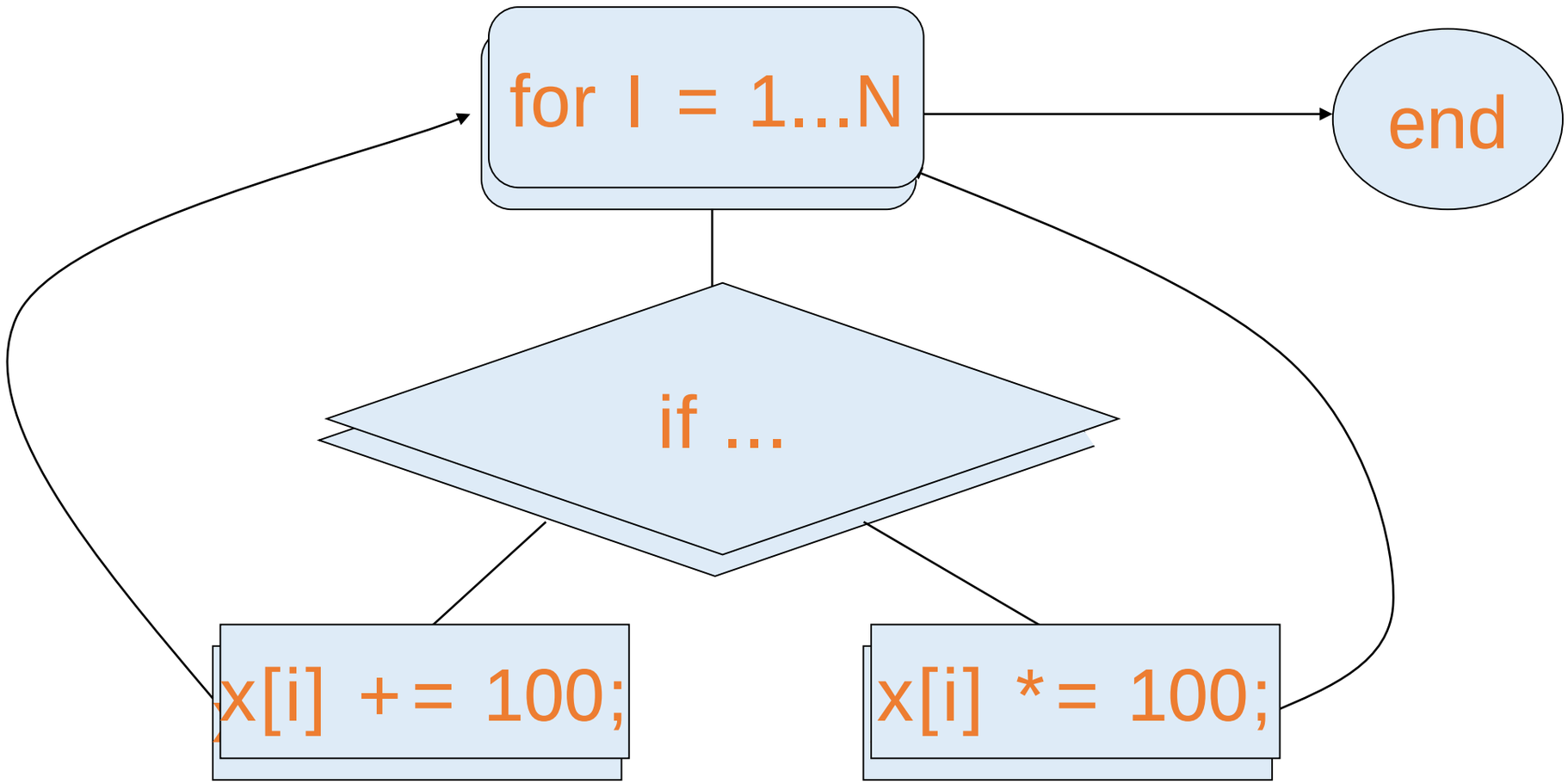
- Line coverage
- Statement
- Function/Method coverage
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- **Path coverage**
- **Loop coverage**
- Mutation adequacy
- …

Path Coverage

Adequacy criterion: each path must be executed at least once

Coverage:

$$\frac{\text{\# executed paths}}{\text{\# paths}}$$



Branch vs Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How many test cases to achieve branch coverage?

Two, for example:

1. **cond1: true, cond2: true**
2. **cond1: false, cond2: false**

Branch vs Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How about path coverage?

Four:

1. **cond1: true, cond2: true**
2. **cond1: false, cond2: true**
3. **cond1: true, cond2: false**
4. **cond1: false, cond2: false**

Branch vs Path Coverage

```
if( cond1 )
   f1();
else
   f2();

if( cond2 )
   f3();
else
   f4();

if( condN )
   fN();
else
   fN();

if( condN )
   fN();
else
   fN();
if( condN )
   fN();
if( condN )
   fN();
if( condN )
   fN();
if( condN )
   fN();
else
   fN();
```

How many test cases for path coverage?

2^n test cases

Path Coverage

- "Loop boundary" testing:
  - Limit the number of traversals of loops: Zero, once, many
- "Boundary interior" testing:
  - Unfold loop as tree
- "Linear Code Sequence and Jump", LCSJ
  - Limit the length of the paths to be traversed
- "Cyclomatic complexity" / McCabe
  - "Linearly independent paths"

Is 100% Coverage *Feasible*?

- Mutually exclusive conditions
  - (a < 0 && a < 10)
    - (**T && F)** is not feasible
- Dead code / unreachable code
- “This should never happen” code

Systems in practice:
Statement coverage 85-90% feasible

Infeasible Paths Example

```
int example (int a) {
  int r = OK;
  int a = -1;

  if(a == -1) {
    r = ERROR_CODE;
    ERXA_LOG(r);
  }

  if(a == -2) {
    r = OTHER_ERROR_CODE;
    ERXA_LOG(r);
  }

  return r;
}
```

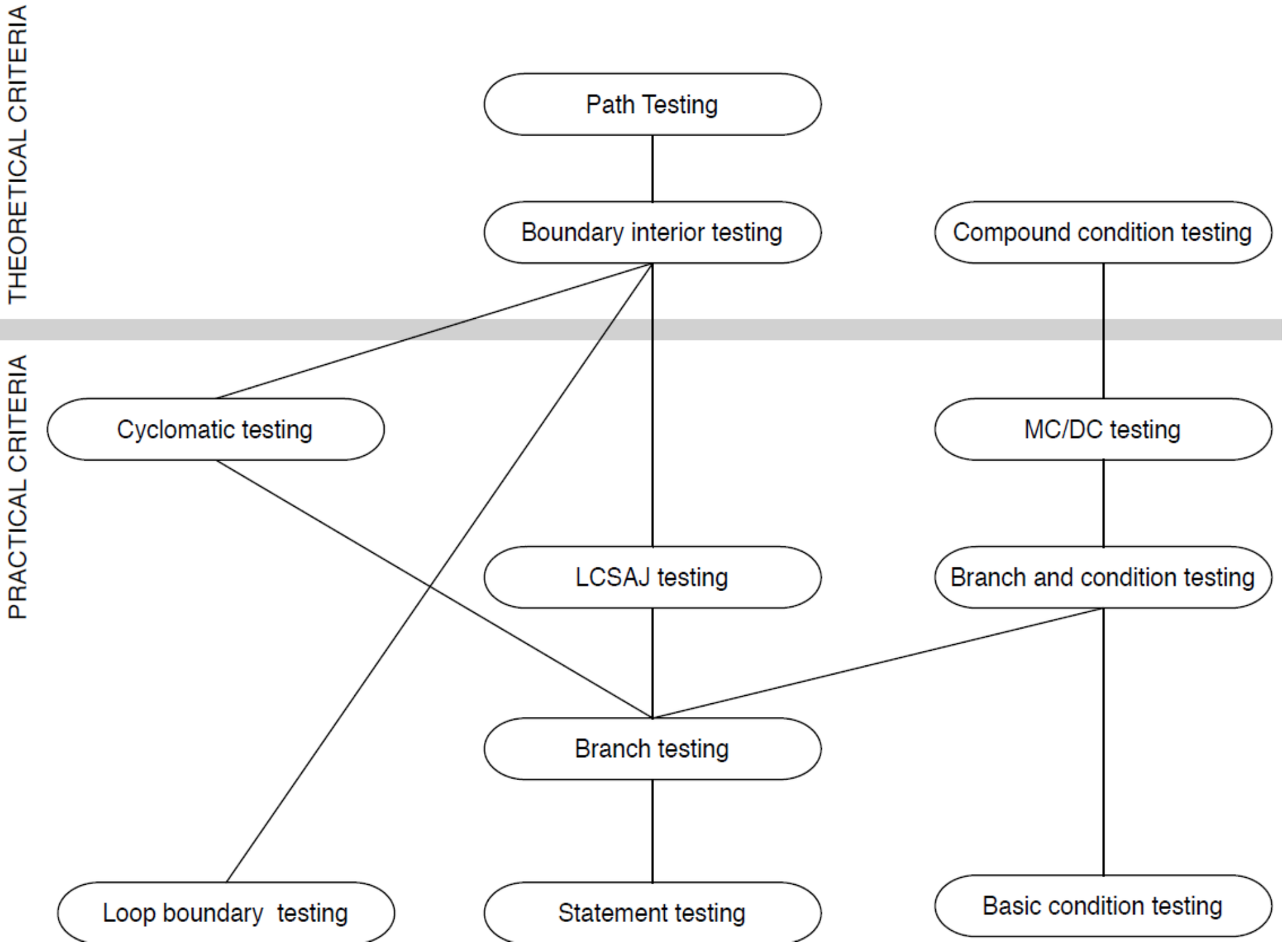
Three feasible paths: a = -1; a = -2 or any other a.
Infeasible path: a == -1 as well as a == -2.

Coverage Strategy Subsumption

- Strategy X **subsumes** strategy Y if
  - all elements that Y covers are also covered by X

- Subsumes hierarchy:
  - Analytical ranking of coverage metrics
  - E.g.: MC/DC > all branches > all statements

Subsumption relation



Coverage: Useful or Harmful?

- Measuring coverage (% of satisfied test obligations) can be a useful indicator ...
  - Of progress toward a thorough test suite, of trouble spots requiring more attention
- ... or a dangerous seduction
  - Coverage is only a proxy for thoroughness or adequacy
  - It's easy to improve coverage without improving a test suite (much easier than designing good test cases)

- The only measure that really matters is **effectiveness**

Deductive Reasoning by Example

Example: a library information system

- Consider a (very) simplified library, where new books can be added to and removed from the library's holdings, and users can borrow and return books.
- Some examples of specifications for this domain:
  - *No matter what the state of the library is, every book in the library's holdings must be available to be borrowed, or be on loan.*

$$\forall b,s[InHoldings(b,s) \quad Available(b,s) \vee \exists p\ OnLoan(b,p,s)]$$

Here, S is a state variable.

- *In every state, no book may be simultaneously on loan and available to be borrowed.*

$$\forall b,s\{InHoldings(b,s) \Rightarrow \neg[Available(b,s) \wedge \exists p\ OnLoan(b,p,s)]\}$$

- *Two different people cannot have the same book on loan.*

$$\forall b,s\ \{InHoldings(b,s) \Rightarrow \neg\exists p,p'[OnLoan(b,p,s) \wedge\ OnLoan(b,p',s) \wedge p \neq p']\}$$

System transactions

- Operations that change the state of the system are called transactions.

  For the library system,

- a person p borrowing book b

  is such a transaction, whose effect is to change the current state s of the library to a state s' in which $OnLoan(b,p,s')$ and $\neg Available(b,s')$ hold.

- Transactions have preconditions, which are conditions on the system state under which it is possible to perform the transaction.
- Transactions have postconditions, which are the properties that must be true in the system state resulting from performing the transaction.
- We specify a transaction by specifying its precondition and postcondition.

Invariance under transactions

- What kinds of properties do we want to prove?
- The most important are system properties that are *invariant under transactions*, defined as follows:

> When property P holds in system state s, and a transaction changes the system state from s to s', then P continues to hold in the new state s'.

- So for a property P that is invariant, assuming that P holds in the initial state of the system, P will be true in *every* state that the system passes through.

  That is, no matter what transactions are performed, and no matter how many of them are performed, nothing will ever make P false.

- Why is this good?

  For an air traffic control system, consider the property:

  No two planes are on an arrival runway within a one minute time period.

Invariance as an entailment

- Now assume that the programmer has done his/her job correctly. This means we have the following:

  If the precondition holds in s, and the transaction changes the system from s to s', then the postcondition holds from s to s',

- To show invariance, it is therefore sufficient to prove:

  If P holds in s, and the precondition holds in s, and the postcondition holds from s to s', then P will hold in s'.

- Formally, we wish to prove that

$$\Sigma \models \forall\, x_1,\ldots,x_n,s,\, s'\ (P(s) \wedge \mathit{TransPrecond}(x_1,\ldots,x_n,s) \wedge \mathit{TransPostcond}(x_1,\ldots,x_n,s,s') \supset P(s'))$$

where
- Σ defines the transaction pre and postconditions
- $x_1,\ldots,x_n$ are the arguments of the transaction
- P is the property to be shown invariant
- $\mathit{TransPrecond}$ is the transaction precondition
- $\mathit{TransPostcond}$ is the transaction postcondition

Showing invariants for the library

Aim to show that the first desired property
$\forall b\ [InHoldings(b,s) \Rightarrow\ Available(b,s) \vee \exists p\ OnLoan(b,p,s)]$

is invariant under the transaction of borrowing a book.

In other words, we prove the following:

$\Sigma \models \forall\ b,p,s,s'\ (P(s) \wedge BorrowPrecond(b,p,s) \wedge BorrowPostcond(b,p,s,s') \Rightarrow P(s'))$,

where $P(s)$ is the property above, and where Σ contains the equivalences $BorrowPrecond$ and $BorrowPostcond$.

Proof sketch

We sketch the structure of a derivation, starting with the entailment to be proved as the root:

1. Using $\forall\mathbf{I}$ and $\supset\mathbf{I}$ repeatedly, replace universals by new constants and move antecedents into Σ:

$$\Sigma' \models Available(\beta,\sigma') \vee \exists p\ OnLoan(\beta,p,\sigma')$$

Greek letter = new constant

2. Using the derived rule for equivalences (see below), expand the *BorrowPrecond* and *BorrowPostcond* in Σ. Then use the equivalences from the postcondition to eliminate all of the σ' arguments to predicates:

$$\Sigma'' \models (Available(\beta,\sigma) \wedge \beta \neq \gamma) \vee \exists p\ (\beta=\gamma \ \wedge\ p=\rho \ \vee\ OnLoan(\beta,p,\sigma)))$$

3. Use logical equivalences (see below) to reformulate the conclusion in the following equivalent form:

$$\Sigma'' \models Available(\beta,\sigma) \ \vee\ \beta=\gamma \vee\ \exists p\ OnLoan(\beta,p,\sigma)$$

4. Use the rule $\vee\mathbf{I}$ to reduce this to:

$$\Sigma''' \models Available(\beta,\sigma) \ \vee\ \exists p\ OnLoan(\beta,p,\sigma)$$

5. Reduce this to two goals using $\supset\mathbf{E}$:

$$\Sigma''' \models InHoldings(\beta,\sigma)$$

$$\Sigma''' \models InHoldings(\beta,\sigma) \supset (Available(\beta,\sigma) \ \vee\ \exists p\ OnLoan(\beta,p,\sigma))$$

6. The first goal is solved immediately in Σ'''. The second is solved by applying $\forall\mathbf{E}$ (to the initial invariant in Σ''').

Using equivalences

- The use of equivalences is very important in proofs. In the above proof, we used two
- derived rules of inference for equivalences:

  - **IF** $\Sigma \models \forall x_1 \ldots x_n (P(x_1,\ldots,x_n) \equiv H)$ and $\Sigma \models F$ **THEN** $\Sigma' \models F'$

  where F' and Σ' are like F and Σ, but with some occurrences of $P(t_1,\ldots,t_n)$ replaced by $H[x_1/t_1,\ldots,x_n/t_n]$

  This derived rule is most often used when a "definition" of P is included as part of Σ.

  The equivalences that were used in Step 2:

  - $\Sigma \models \forall p \forall b \forall s$ (*BorrowPrecond(p,b,s)* ≡ **...)**
  - $\Sigma \models \forall p \forall b \forall s \forall s'$ (*BorrowPostcond(p,b,s,s′)* ≡ **...)**
  - $\Sigma \models \forall b'$(*Available(b′,σ′)* ≡ **...)** (from the postcondition)
  - $\Sigma \models \forall b'$(*InHoldings(b′,σ′)* ≡ **...)** (from the postcondition)
  - $\Sigma \models \forall b' p'$(*OnLoan(b′,p′ σ′)* ≡ **...)** (from the postcondition)

- **IF** $\Sigma \models (G \equiv H)$ and $\Sigma \models F$ **THEN** $\Sigma' \models F'$

where F' and Σ' are like F and Σ, but with some occurrences of G as a subformula replaced by H.

This derived rule is most often used when H is *logically equivalent* to but simpler than G. A separate subproof is usually needed to prove this equivalence.

The equivalences that were used in Step 3 of the proof are on the next slide.

Equivalences used in the invariant proof

- In Step 3 of the proof above, a number of logical equivalences were used to reformulate the conclusion:

$$(Available(\beta,\sigma) \wedge \beta \neq \gamma) \vee \exists p\ (\beta=\gamma\ \wedge\ p=\rho\ \vee\ OnLoan(\beta,p,\sigma)))$$

use equivalence: $\exists x(F \vee G) \equiv \exists xF \vee \exists xG$

$$(Available(\beta,\sigma) \wedge \beta \neq \gamma) \vee \exists p\ (\beta=\gamma\ \wedge\ p=\rho)\ \vee\ \exists p\ OnLoan(\beta,p,\sigma))$$

then use equivalence: $\exists x(F \wedge G) \equiv F \wedge \exists xG$, when x is not free in F

$$(Available(\beta,\sigma) \wedge \beta \neq \gamma) \vee (\beta=\gamma\ \wedge\ \exists p\ (p=\rho))\ \vee\ \exists p\ OnLoan(\beta,p,\sigma))$$

then use equivalence: $F \wedge \exists x(x=c) \equiv F$

$$(Available(\beta,\sigma) \wedge \beta \neq \gamma) \vee \beta=\gamma\ \vee\ \exists p\ OnLoan(\beta,p,\sigma)$$

then use equivalence: $(F \wedge \neg G) \vee G \equiv F \vee G$

$$Available(\beta,\sigma)\ \vee\ \beta=\gamma\ \vee\ \exists p\ OnLoan(\beta,p,\sigma)$$

Computer-aided theorem-proving

- Even for the simple library system, proving the invariant properties by hand is lengthy and tedious.
- Fortunately, much of this work can be automated.

  The rules of inference, plus their derived rules, can be implemented.
- The resulting theorem-proving system can be fully automatic, or interactive.
  - In the first case, the system is expected to find a derivation on its own. This is feasible for simple theorems like those for the library.

    cf. systems such as Otter, SETHEO
  - But many theorems require human ingenuity and insight, in which case interactive theorem-provers perform the "routine" work, interactively asking the user for help when they get stuck.

    cf. systems such as PVS, NuPerl

Summary: Validation of Systems

The principal methods for the validation of complex parallel and distributed systems are:

- Testing (using the system itself)
- Simulation (using a model of the system)
- Deductive verification (mathematical (manual) proof of correctness, in practice done with computer aided proof assistants/theorem provers)
- Model Checking (≈ exhaustive testing of a model of the system)