

## CTL Model checking

- Assumptions:
  1. finite number of processes, each having a finite number of finite-valued variables.
  2. finite length of CTL formula
- Problem: Determine whether formula  $f_0$  is true in a finite structure  $M$ .
- Algorithm overview:
  1.  $f_0 = \text{TRANSLATE}(f_0)$  (in terms of AF, EU, EX,  $\wedge$ ,  $\vee$ ,  $\perp$ )
  2. Label the states of  $M$  with the subformulas of  $f_0$  that are satisfied there and work outwards towards  $f_0$ .  
Ex:  $\text{AF}(a \wedge \text{E}(b \cup c))$
  3. If starting state  $s_0$  is labeled with  $f_0$ , then  $f_0$  holds on  $M$ , i.e.

$$(s_0 \in \{s \mid M, s \models f_0\}) \Rightarrow (M \models f_0)$$

39

## Model-Checking

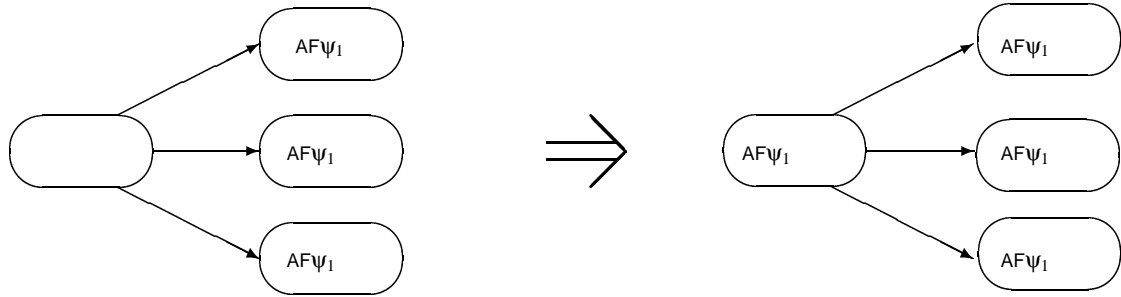
- Idea of model-checking: establish that the system is a model of a formula (doing a search).
- CTL Model Checking
- SMV input language and its semantics
- SMV examples
- Model checking with fairness
- Binary Decision Diagrams.
- Symbolic model-checking and fixpoints.

38

## Labeling Algorithm (Cont'd)

- $AF \psi_1$ :
  - If any state  $s$  is labeled with  $\psi_1$ , label it with  $AF \psi_1$ .
  - Repeat: label any state with  $AF \psi_1$  if all successor states are labeled with  $AF \psi_1$ , until there is no change.

Ex:



41

## Labeling Algorithm

Suppose  $\psi$  is a subformula of  $f$  and states satisfying all the immediate subformulas of  $\psi$  have already been labeled. We want to determine which states to label with  $\psi$ . If  $\psi$  is:

- $\perp$ : then no states are labeled with  $\perp$ .
- $p$  (prop. formula): label  $s$  with  $p$  if  $p \in I(s)$ .
- $\psi_1 \wedge \psi_2$ : label  $s$  with  $\psi_1 \wedge \psi_2$  if  $s$  is already labeled both with  $\psi_1$  and with  $\psi_2$ .
- $\neg\psi_1$ : label  $s$  with  $\neg\psi_1$  if  $s$  is not already labeled with  $\psi_1$ .
- $EX \psi_1$ : label any state with  $EX \psi_1$  if one of its successors is labeled with  $\psi_1$ .

40

## Handling $EG\psi_1$ directly

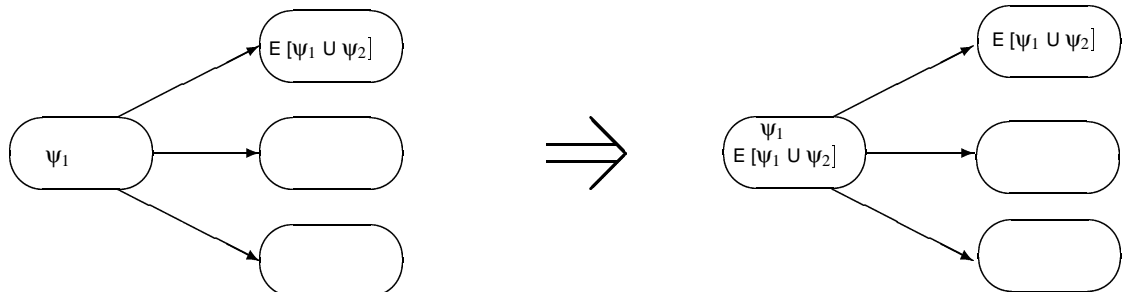
- $EG\psi_1$ :
  - Label *all* the states with  $EG\psi_1$ .
  - If any state  $s$  is *not* labeled with  $\psi_1$ , *delete* the label  $EG\psi_1$ .
  - Repeat: *delete* the label  $EG\psi_1$  from any state if *none* of its successors is labeled with  $EG\psi_1$ ; until there is no change.

43

## Labeling Algorithm (Cont'd)

- $E[\psi_1 \cup \psi_2]$ :
  - If any state  $s$  is labeled with  $\psi_2$ , label it with  $E[\psi_1 \cup \psi_2]$ .
  - Repeat: label any state with  $E[\psi_1 \cup \psi_2]$  if it is labeled with  $\psi_1$  and at least one of its successors is labeled with  $E[\psi_1 \cup \psi_2]$ , until there is no change.

Ex:



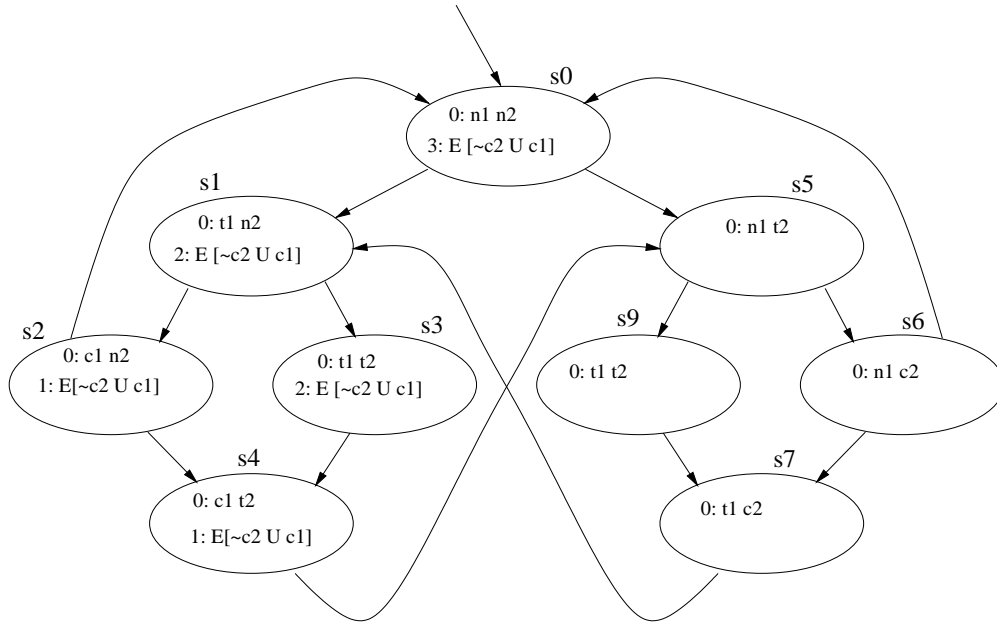
Output states labeled with  $f$ .

Complexity:  $O(|f| \times S \times (S + |R|))$  (linear in the size of the formula and quadratic in the size of the model).

42

## Example

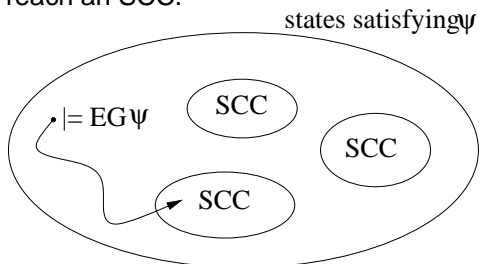
Verifying  $E[\neg c_2 \text{ U } c_1]$  on the mutual exclusion example.



45

## Even Better Handling of EG

- restrict the graph to states satisfying  $\psi_1$ , i.e., delete all other states and their transitions;
- find the maximal *strongly connected components* (SCCs); these are maximal regions of the state space in which every state is linked with every other one in that region.
- use breadth-first searching on the restricted graph to find any state that can reach an SCC.

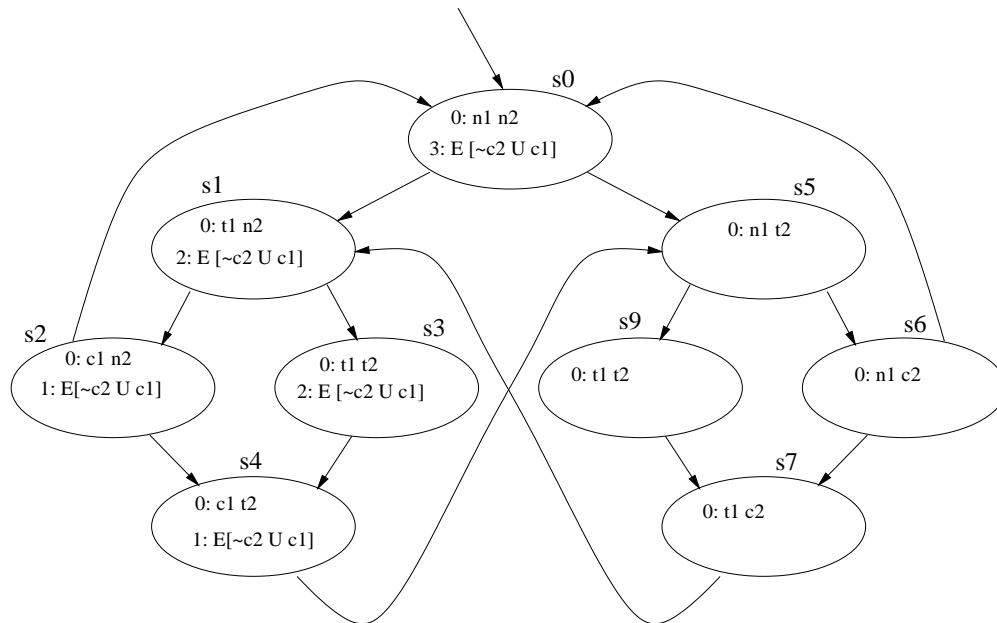


Complexity:  $O(|f| \times (S + |R|))$  (linear in size of model and size of formula).

44

## Example

Verifying  $E[\neg c_2 \text{ U } c_1]$  on the mutual exclusion example.



47

## CTL Model-Checking

- Michael Browne, CMU, 1989.
- Usually for verifying concurrent *synchronous* systems (hardware, SCR specs...)
- Specify correctness criteria: safety, liveness...
- Instead of keeping track of labels for each state, keep track of a set of states in which a certain formula holds.

46

## Generating Counterexamples

Only works for universal properties

- $AXp$
- $AG(p \Rightarrow AFq)$
- etc.

Step 1: negate the property and express it using  $EX$ ,  $EU$ , and  $EG$

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

Step 2:

- For  $EXp$  – find a successor state labeled with  $p$
- For  $EGp$  – follow successors labeled with  $EGp$  until a loop is found
- For  $E[pUq]$  – remove all states not labeled with  $p$  or  $q$ , then look for path to  $q$

49

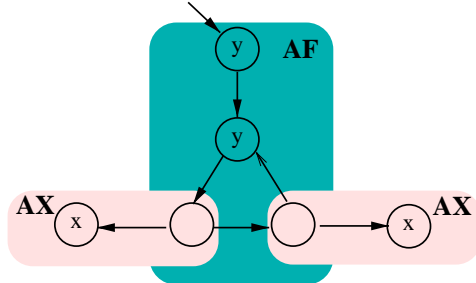
## Counterexamples and Witnesses

- Counterexamples
  - explains why a property is false
  - typically a violating path for universal properties
  - how to explain that something does not exist?
- Witnesses
  - explains why a property is true
  - typically a satisfying path for existential properties
  - how to explain that something holds on all paths?

48

## Are counterexamples always linear?

- SMV only supports linear counterexamples
- But what about  $(AXp) \vee (AXq)$ ?
- Counterexample for  $AF(\neg y \wedge AX\neg x)$



- See: E. Clarke et al. “Tree-Like Counterexamples in Model Checking”, Proceedings of LICS’02.

51

## Counterexamples and Witnesses (Cont’d)

- What about properties that combine universal and existential operators?
- Are they really different?
  - a counterexample for  $\varphi$  is a witness to its negation
  - a counterexample for a universal property is a witness to some existential property
  - e.g.  $AGp$  and  $EF\neg p$
- One alternative
  - build a proof instead of a counterexample
  - works for all properties (but proofs can be big)
  - see:
    - \* A. Gurfinkel and M. Chechik. “Proof-like Counterexamples”, Proceedings of TACAS’03.
    - \* M. Chechik, A. Gurfinkel. “A Framework for Counterexample Generation and Exploration”, FASE’2005.

50

## SMV

Symbolic model verifier – a model-checker that uses symbolic model checking algorithm. The language for describing the model is a simple parallel assignment.

- Can have synchronous or asynchronous parallelism.
- Model environment non-deterministically.
- Also use non-determinism for systems which are not fully implemented or are abstract models of complex systems.

53

## State Explosion

Imagine that you a Kripke structure of size  $n$ . What happens if we add another boolean variable to our model?

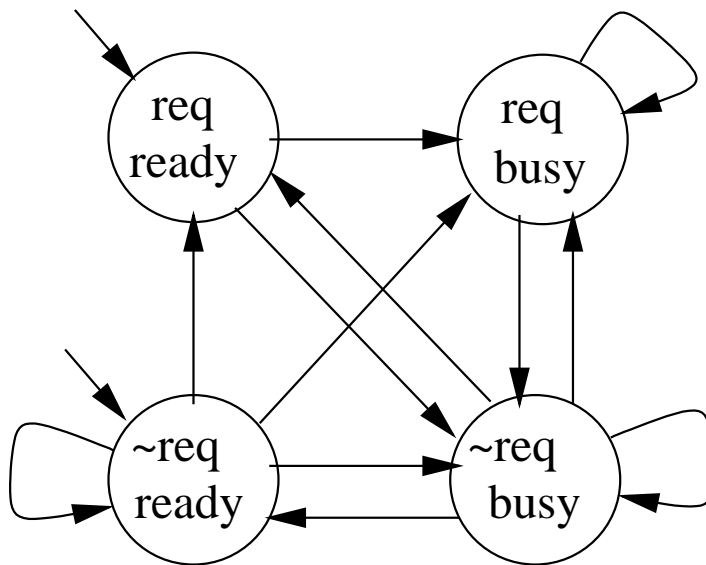
How to deal with this problem?

- Symbolic model checking with efficient data structures (BDDs). Don't need to represent and manipulate the entire model. Model-checker SMV [McMillan, 1993].
- Abstraction: we abstract away variables in the model which are not relevant to the formula being checked (see later in the course).
- 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.

52



### Model for First SMV Example



55

### First SMV Example

```
MODULE main
VAR
    request : boolean;
    state : {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) := case
        request : busy;
        1: {ready, busy}
    esac;
SPEC
    AG(request -> AF state = busy)
```

Note that `request` never receives an assignment – this models input.

54

## Another Example

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
SPEC
  AG AF bit2.carry_out

SPEC AG(!bit2.carry_out)

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := (value + carry_in) mod 2;
DEFINE
  carry_out := value & carry_in;
```

57

## More About the Language

- Program may consist of several modules, but one has to be called `main`.
- Each variable is a state machine, described by `init` and `next`.
- Variables are passed into modules by reference.
- Comment – anything starting with `--` and ending with a newline.
- No loop construct.
- Datatypes: `boolean`, enumerated types, user-defined modules, arrays, integer subranges.

```
VAR
  state : {on, off};
  state1 : array 2..5 of {on, off};
  state2 : computeState(1);
  state3 : compute;
  state4 : array 2..5 of state; <- error
  state5 : array on..off of boolean; <- error
```

56

## Modeling Interleaving

Keyword `process` for modeling interleaving. The program executes a step by non-deterministically choosing a process, then executing all of its assignment statements in parallel.

```
MODULE main
VAR
  gate1 : process inverter(gate3.output);
  gate2 : process inverter(gate1.output);
  gate3 : process inverter(gate2.output);
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := 0;
  next(output) := !input;
```

59

## Notation Used

- $a.b$  – component  $b$  of module  $a$ .
- `DEFINE` – same as `ASSIGN` but
  - cannot be given values non-deterministically
  - is dynamically typed
  - does not increase the size of state space.
  - like `#define` in C

## Fixing the Example

```
MODULE main
VAR
  gate1 : process inverter(gate3.output);
  gate2 : process inverter(gate1.output);
  gate3 : process inverter(gate2.output);
SPEC
  (AG AF gate1.output) & (AG AF !gate1.output)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := 0;
  next(output) := !input;
FAIRNESS
  running
-- specification AG AF gate1.output .. is true
```

61

## Output of Running SMV

```
-- specification AG AF gate1.output & ... is false
-- as demonstrated by the following execution sequence
-- loop starts here --
state 1.1:
gate1.output = 0
gate2.output = 0
gate3.output = 0
[stuttering]

state 1.2:
[stuttering]

resources used:
user time: 0.11 s, system time: 0.16 s
BDD nodes allocated: 303
Bytes allocated: 1245184
BDD nodes representing transition relation: 32 + 1
```

What went wrong? We never specified that each process has to execute infinitely often – a *fairness* constraint.

60

## Mutual Exclusion Again

st – status of the process (critical section, or not, or trying)

other-st – status of the other process

turn – ensures that they take turns

```
MODULE main
  VAR
    pr1 : process prc(pr2.st, turn, 0);
    pr2 : process prc(pr1.st, turn, 1);
    turn : boolean;
  ASSIGN
    init(turn) := 0;
  --safety
  SPEC AG!((pr1.st = c) & (pr2.st = c))
  --liveness
  SPEC AG((pr1.st = t) -> AF (pr1.st = c))
  SPEC AG((pr2.st = t) -> AF (pr2.st = c))
  --no strict sequencing
  SPEC EF(pr1.st = c & E[pr1.st = c U
    (!pr1.st = c & E[! pr2.st = c U pr1.st = c ]))
```

63

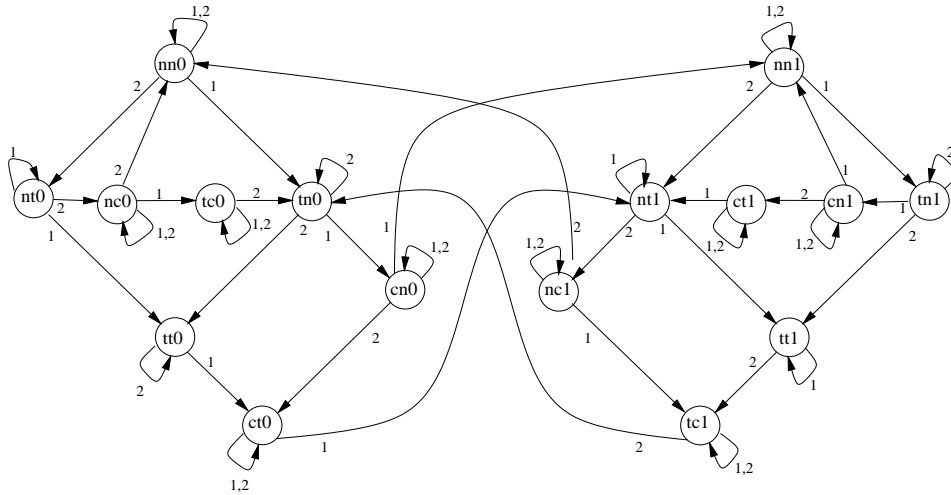
## Advantages of Interleaving Model

- Allows for a particularly efficient representation of the transition relation:

The set of states reachable by one step of the program is the union of the sets reachable by each individual process. So, do not need reachability graph.

- But sometimes have increased complexity in representing the set of states reachable in  $n$  steps (can have up to  $s^n$  possibilities).

## Model



65

## Model (Cont'd)

```

MODULE prc(other-st, turn, myturn)
  VAR
    st : {n, t, c};
  ASSIGN
    init(st) := n;
    next(st) := case
      (st = n) : {t, n};
      (st = t) & (other-st = n) : c;
      (st = t) & (other-st = t) & (turn = myturn) : c;
      (st = c) : {c, n};
    1 : st;
      esac;
    next(turn) := case
      turn = myturn & st = c : !turn;
      1 : turn;
    esac;
  FAIRNESS running
  FAIRNESS !(st = c)

```

64

## Fairness (Again)

Let  $C = \{\psi_1, \psi_2, \dots, \psi_n\}$  be a set of  $n$  fairness constraints. A computation path  $s_0, s_1, \dots$  is *fair* w.r.t.  $C$  if for each  $i$  there are infinitely many  $j$  s.t.  $s_j \models \psi_i$ , that is, each  $\psi_i$  is true infinitely often along the path.

We use  $A_C$  and  $E_C$  for the operators  $A$  and  $E$  restricted to fair paths.

$E_C U$ ,  $E_C G$  and  $E_C X$  form an adequate set.

$E_C G \top$  holds in a state if it is the beginning of a fair path.

Also, a path is fair iff any suffix of it is fair. Finally,

$$E_C[\phi U \psi] = E[\phi U (\psi \wedge E_C G \top)]$$

$$E_C X \phi = EX(\phi \wedge E_C G \top)$$

We only need a new algorithm for  $E_C G \phi$

67

## Comments:

- The labels in the slide above denote the process which can make the move.
- Variable `turn` was used to differentiate between states  $s_3$  and  $s_9$ , so we now distinguish between `ct0` and `ct1`. But transitions out of them are the same.
- Removed the assumption that the system moves on each tick of the clock. So, the process can get stuck, and thus the fairness constraint.
- In general, what is the difference between the single fairness constraint  $\psi_1 \wedge \psi_2 \wedge \dots \wedge \psi_n$  and  $n$  fairness constraints  $\psi_1, \psi_2, \dots$ , written on separate lines under FAIRNESS?

66

## Guidelines for Modeling with SMV

- Identify inputs from the environment.
- Make sure that the environment is non-deterministic. All constraints on the environment should be carefully justified.
- Determine the states of the system and its outputs. Model them in terms of the environmental inputs.
- Specify fairness criteria, if any. Justify each criterium. Remember that you can over-specify the system. Fairness may not be implementable, and in fact may result in no behaviors.
- Specify correctness properties (in CTL or LTL). Comment each property in English.
- Ensure that desired properties are not satisfied vacuously.

69

## Algorithm for $E_C G \phi$

- Restrict the graph to states satisfying  $\phi$ ; of the resulting graph, we want to know from which states there is a fair path.
- Find the maximal *strongly connected components* (SCCs) of the restricted graph;
- Remove an SCC if, for some  $\psi_i$ , it does not contain a state satisfying  $\psi_i$ . The resulting SCCs are the fair SCCs. Any state of the restricted graph that can reach one has a fair path from it.
- Use breadth-first search backward to find the states on the restricted graph that can reach a fair SCC.

Complexity:  $O(n \times |f| \times (S + |R|))$

(still linear in the size of the model and formula).

68



## Sanity Checks

- Check that the model is non-trivial
  - $EX\text{true}$  – at least one successor state
  - $AGEX\text{true}$  – transition relation is total
- If result of model-checking is false, there is a counterexample to prove it. If the result is true, no extra information is given!
- Check that every part of the property matters (vacuity checking).
  - Replace consequent of an implication with false and check
  - If  $AG(p \Rightarrow AFq)$ , check  $AG(p \Rightarrow \text{false})$
  - The result should be false.
  - The counterexample shows one good execution.
- Use counterexamples for simulation.
  - Example:  $\neg EF(\text{floor} = 2)$

71

## Vacuity in Temporal Logic

- Let  $\phi[\psi]$  be a formula with subformula  $\psi$
- $\psi$  *affects*  $\phi[\psi]$  if replacing  $\psi$  with another subformula changes the value of  $\phi$
- $\phi[\psi]$  is *vacuous in*  $\psi$  if  $\psi$  does not affect  $\phi$
- $\phi$  is *vacuous* if there exists a subformula  $\psi$  such that  $\phi$  is vacuous in  $\psi$
- To check if  $\phi[\psi]$  is vacuous in an occurrence of  $\psi$ 
  - check  $\phi[\psi \leftarrow \text{true}]$
  - check  $\phi[\psi \leftarrow \text{false}]$
  - $\phi$  is vacuous if both results are the same
- Further reading
  - I. Beer et al. “Efficient Detection of Vacuity in Temporal Model Checking”, *FMSD*, 2001.
  - O. Kupferman and M. Vardi. “Vacuity Detection in Temporal Model Checking”, *STTT*, 2003.
  - A. Gurfinkel and M. Chechik. “How Vacuous is Vacuous”, TACAS’04.

70