## Model-Checking

Readings: Textbook, 3.4-3.7, 3.9; References [5]-[8]

- Idea of model-checking: establish that the system is a model of a formula (doing a search).
- CTL and Kripke structures
- CTL Model Checking
- SMV input language and its semantics
- SMV examples
- Notion of fairness
- Introduction to BDDs
- Symbolic model-checking and fixpoints.
- State of the art model-checkers.



## CTL

Computational tree logic - propositional branching time logic, permitting explicit quantification over all possible futures.

Syntax:
1. Every atomic proposition is a CTL formula
2. If $f$ and $g$ are CTL formulae, then so are $\neg f$, $f \wedge g$, $f \vee g$, AX$f$, EX$f$, A[$f$U$g$], E[$f$U$g$], AF$f$, EF$f$, AG$f$, EG$f$.

Temporal operators - quantifier (A or E) followed by F (future), G (global), U (until), or X (next).



## Where did the "tree" come from?

Fig. 1, page 247 of TOPLASS paper



## CTL - Cont'd

Only X and U are necessary. The rest are abbreviations:

$$
\begin{array}{lcl}
(f \wedge g) & \equiv & \sim(\sim f \vee \sim g) \\
(f \rightarrow g) & \equiv & (\sim f \vee g) \\
\mathsf{AX}f & \equiv & \sim\mathsf{EX}(\sim f) \\
\mathsf{EF}f & \equiv & \mathsf{E}[\mathsf{true}\ \mathsf{U}\ f] \\
\mathsf{AF}f & \equiv & \mathsf{A}[\mathsf{true}\ \mathsf{U}\ f] \\
\mathsf{EG}f & \equiv & \sim\mathsf{AF}(\sim f) \\
\mathsf{AG}f & \equiv & \sim\mathsf{EF}(\sim f)
\end{array}
$$

## Kripke Structure

Formula is defined with respect to a model $M = \langle V, S, s_0, R, I \rangle$, where

$V$ is a set of propositional variables

$S$ is a set of states

$s_0 \in S$ is the start state

$R$ is a transition relation (every state has a successor)

$I$ is a set of interpretations specifying which propositions are true in each state.

Temporal logic formulas are evaluated with respect to a state in the model:

EX($f$) (AX($f$)) is true in $M, s_i$ if $f$ is true in some (all) successor of $M, s_i$, etc.

Given a formula $f$, we want to know if $M, s_0 \models f$.



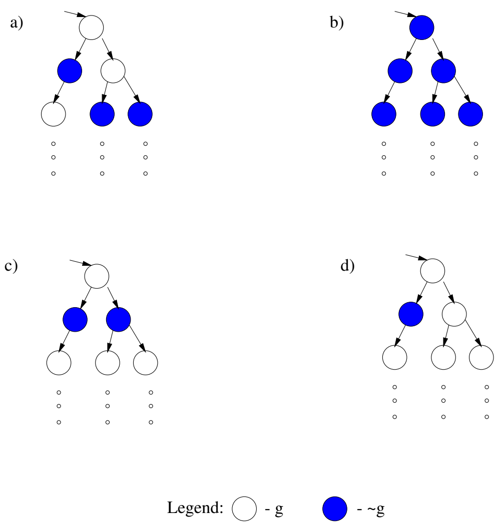
## Formally...

| | | |
|---|---|---|
| $M, s \models p$ | iff | $(I(s))(p)$ |
| $M, s \models \neg f$ | iff | $s \not\models f$ |
| $M, s \models f \vee g$ | iff | $s \models f$ or $s \models g$ |
| $M, s_0 \models$ EX$\neg f$ | iff | $\exists$ path $(s_0, s_1, \ldots)), s_1 \models \neg f$ |
| $M, s_0 \models$ AX$\neg f$ | iff | $\forall$ paths $(s_0, s_1, \ldots)), s_1 \models \neg f$ |
| $M, s_0 \models$ E($f$U$g$) | iff | $\exists$ path $(s_0, s_1, \ldots)),$ |
| | | $\exists i$ s.t. $M, s_i \models g$ and $\forall j < i, s_j \models f$ |
| $M, s_0 \models$ A($f$U$g$) | iff | $\forall$ paths $(s_0, s_1, \ldots)),$ |
| | | $\exists i$, s.t. $M, s_i \models g$ and $\forall j < i, s_j \models f$ |



## Examples

What formulas hold in these models?






## Specifications using CTL

Thermostat.

1. Temperature is always above normal, below normal or normal.

2. When temperature becomes above normal, in the following state the AC will be turned on.

3. Thermostat is never running heater and AC at the same time.

## 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 the finite structure $M$.

Algorithm overview:
1. $f_0 = \texttt{TRANSLATE}(f_0)$ (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 working outwards towards $f_0$.

Ex: AF($a$ $\wedge$ E($b$ U $c$))

3. If starting state $s_0$ is in the final set, then $f_0$ is holds on $M$, i.e.

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



## 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 $\psi_1$.

- 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.



## Labeling Algorithm (Cont'd)

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

- EX $\psi_1$: label any state with EX $\psi_1$ if one of its successors is labeled with $\psi_1$.

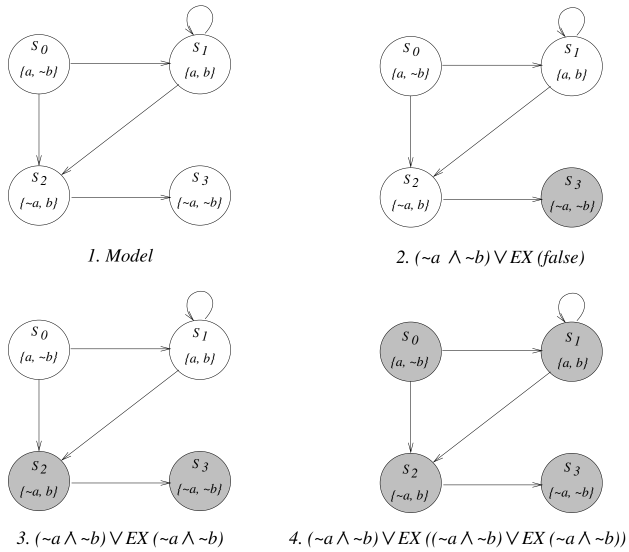
Output states labeled with $f$.

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



## Example

Model checking $M, s_0 \models$ E($a$U$\neg b$)




1. Model

2. (~a ∧ ~b) ∨ EX (false)

3. (~a ∧ ~b) ∨ EX (~a ∧ ~b)

4. (~a ∧ ~b) ∨ EX ((~a ∧ ~b) ∨ EX (~a ∧ ~b))

## Handling EG$\psi_1$ directly

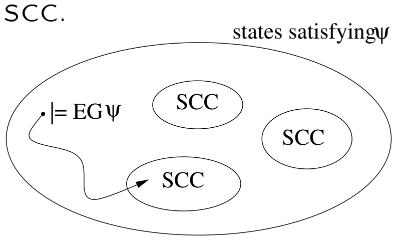
• EG $\psi_1$:
- Label *all* the states with EG $\psi_1$.
- If any state $s$ is *not* labeled with $\psi_1$, *delete* the label EG $\psi_1$.
- Repeat: *delete* the label EG $\psi_1$ from any state if *none* of its successors is labeled with EG $\psi_1$; until there is no change.

This is a *backward* analysis.



## 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).



## 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 (see 3.5.2 of textbook).



## 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. See Ch. 6 and later in the course. Model-checker SMV [McMillan, 1993].

• Abstraction: we abstract away variables in the model which are not relevant to the formula being checked (this is what we did in SCR).

• 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.

## SMV

Symbolic model verifier – a program 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.
• Will talk about techniques later.



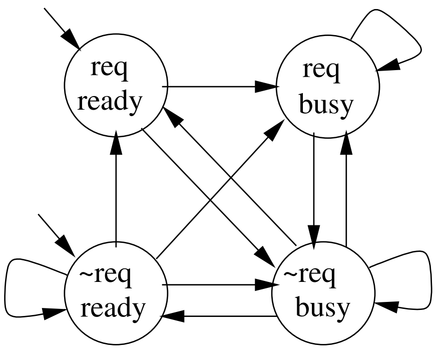
## First SMV Example

```
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.



## Model for First SMV Example






## 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 loops.
• 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
```

## 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;
```

- *a.b* – component $b$ of module $a$.
- `DEFINE` – same as `ASSIGN` but
  - cannot be given values non-deterministically
  - are dynamically typed
  - do not increase the size of state space.



## Models of Concurrency

Maximum parallelism – "simultaneous execution of atomic actions in all system modules capable of performing an operation."

Interleaving – "concurrent execution of modules is represented by interleaving of their atomic actions".

Example:



## 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;
```



## 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.

## 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

resources used:
user time: 0.03 s, system time: 0.23 s
BDD nodes allocated: 288
Bytes allocated: 1245184
BDD nodes representing transition relation: 32 + 1
```



## 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).



## Mutual Exclusion

Each process has four states: `idle`, `entering`, `critical` and `exiting`. If variable `semaphore` is zero, it goes to `critical` and sets `semaphore` to one. On exiting the critical region, it sets `semaphore` to zero again.

Mutual exclusion (safety) property:
"two processes cannot be in critical section at the same time"

```
AG (!proc1.state = critical &
   proc2.state = critical)
```



## Mutual Exclusion (Cont'd)

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);
ASSIGN
  init(semaphore) := 0;
SPEC
  AG !(proc1.state = critical & proc2.state = critical)

MODULE user(semaphore)
VAR
  state : {idle, entering, critical, exiting};
ASSIGN
  init(state) := idle;
  next(state) := case
     state = idle : {idle, entering};
     state = entering & !semaphore : critical;
     state = critical : {critical, exiting};
     state = exiting : idle;
     1 : state;
  esac;
  next(semaphore) := case
     state = entering : 1;
     state = exiting : 0;
     1 : semaphore;
  esac;
FAIRNESS
  running
```

## Is the Spec Correct?

```
-- specification AG (!(proc1.state = critical &
   proc2.sta... is true

resources used:
user time: 0.09 s, system time: 0.13 s
BDD nodes allocated: 560
Bytes allocated: 917504
BDD nodes representing transition relation: 69 + 1
```

Let's add a "lack of starvation" criteria:

```
AG (proc1.state = entering -> AF proc1.state = critical)
```

Output now:

```
-- specification AG (proc1.state = entering ->
   AF proc1.s... is false
-- as demonstrated by the following execution sequence
state 1.1:
semaphore = 0
proc1.state = idle
proc2.state = idle

state 1.2:
[executing process proc1]

-- loop starts here --
state 1.3:
proc1.state = entering
```



## Output (Cont'd)

```
state 1.4:
[executing process proc2]

state 1.5:
[executing process proc2]
proc2.state = entering

state 1.6:
[executing process proc1]
semaphore = 1
proc2.state = critical

state 1.7:
[executing process proc2]

state 1.8:
[executing process proc2]
proc2.state = exiting

state 1.9:
semaphore = 0
proc2.state = idle

resources used:
user time: 0.11 s, system time: 0.11 s
BDD nodes allocated: 1421
Bytes allocated: 917504
BDD nodes representing transition relation: 69 + 1
```



## Attempt to Fix the Spec

- Introduce a variable `turn`
- `turn` would ensure that processes take turns entering the critical section.

Spec:

```
MODULE main
VAR
  semaphore : boolean;
  turn : boolean;
  proc1 : process user(semaphore, 1, turn);
  proc2 : process user(semaphore, 0, turn);
ASSIGN
  init(semaphore) := 0;
  init(turn) := 1;
SPEC
-- fairness
  AG (proc1.state = entering -> AF proc1.state = critical)
SPEC
-- safety
  AG !(proc1.state=critical & proc2.state=critical)

MODULE user(semaphore, number, turn)
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
```



## Fixed Spec (Cont'd)

```
    case
      state = idle : {idle,entering};
      state = entering & !semaphore&(turn=number):critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
      1 : state;
    esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
      1 : semaphore;
    esac;
  next(turn) :=
    case
      state = exiting : !turn;
      1 : turn;
    esac;
FAIRNESS
  running
FAIRNESS
  turn = number
```

This spec is correct!

## Food for Thought

In general, what is the difference between the single fairness constraint $\psi_1 \wedge \psi_2 \wedge ... \wedge \psi_n$ and $n$ fairness constraints $\psi_1$, $\psi_2$, etc., written on separate lines under `FAIRNESS`? Write an SMV program with a fairness constraint $a\&b$ which is not equivalent to the two fairness constraints $a$ and $b$. (You can do it in four lines in SMV.)



## TRANS relation

Parallel assignment should be suitable to most purposes. Still, it is possible to specify the transition relation directly as a propositional formula in terms of the current and next values of state variables.
- Any current/next state pair is in the transition relation iff the value of the formula is one.
- It is possible to give the set of possible initial states as a formula in terms of only the current state variables.
- It is done using TRANS and INIT.



## Inverter using INIT and TRANS

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

MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = 0
TRANS
  next(output) = !input |
  next(output) = output
```

Actually, this spec is wrong! Try to fix it!



## Notion of Fairness

*Fairness*: a path $p$ is *fair* w.r.t. property $\psi$ if $\psi$ is true on $p$ infinitely often.

We may want to evaluate A and E constraints only over those paths.

Example: each process will run infinitely often; a process can stay in a critical section arbitrarily long, as long as it eventually leaves.

Two types of fairness: simple
*Property $\phi$ is true infinitely often.*
and compound
*If $\phi$ is true infinitely often, then $\psi$ is also true infinitely often.*

SMV can deal only with simple fairness.

## Formal Definition of Fairness

Let $C = \{\psi_1, \psi_2, ..., \psi_n\}$ be a set of $n$ fairness constraints. A computation path $s_0, s_1, ...$ 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 $\mathsf{A}_C$ and $\mathsf{E}_C$ for the operators A and E restricted to fair paths.

$\mathsf{E}_C\mathsf{U}$, $\mathsf{E}_C\mathsf{G}$ and $\mathsf{E}_C\mathsf{X}$ form an adequate set.

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

$$\mathsf{E}_C[\phi \mathsf{U} \psi] = \mathsf{E}[\phi \mathsf{U}(\psi \wedge \mathsf{E}_C\mathsf{G}\top)]$$

$$\mathsf{E}_C\mathsf{X}\phi = \mathsf{EX}(\phi \wedge \mathsf{E}_C\mathsf{G}\top)$$

We only need an algorithm for $\mathsf{E}_C\mathsf{G}\phi$.



## Algorithm for $\mathsf{E}_C\mathsf{G}\phi$

• Restrict the graph to states satisfying $\psi$; 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 backwards breadth-first searching 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).



## 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 in CTL. Comment each CTL property in English.

• Ensure that CTL properties are not satisfied vacuously. That is, each universally-quantified property should be paired up with an existentially-specified property. Also check that LHSs of implications are not always false.

Examples:
  AG $(a)$ —
  AG $(a \rightarrow b)$ —



## Symbolic model checking

Why?
Saves is from constructing a model's state space. Effective "cure" for state space explosion problem.

How?
Sets of states and transition relations are represented by formulas, and set operations are defined in terms of formula manipulations.

Data structures
BDDs - allow for efficient storage and manipulation of logic formulas.

## BDDs

Insert Andersen's slides here.



## Digression: Fixpoint Computation

Definition: Let $S$ be a set of states and $F : \mathcal{P}(\mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S})$ a function on the power set of $S$.
1. We say that $F$ is *monotone* if $X \subseteq Y$ implies $F(X) \subseteq F(Y)$ for all subsets $X$ and $Y$ of $S$.
2. A subset $X$ of $S$ is called a *fixed point (fixpoint)* of $F$ if $F(X) = X$.

Example: Let $S = \{s_0, s_1\}$ and $F(Y) = Y \cup \{s_0\}$ for all subsets $Y$ of $S$. $F$ is monotone, because $Y \subseteq Y\prime$ implies $Y \cup \{s_0\} \subseteq Y\prime \cup \{s_0\}$.
$F$ has two fixpoints: $\{s_0\}$ (least) and $\{s_0, s_1\}$ (greatest).

Example: Let
$G(Y) =$ if $Y = \{s_0\}$ then $\{s_1\}$ else $\{s_0\}$
$G$ is not monotone (e.g., $\{s_0\} \subseteq \{s_0, s_1\}$, but $G(\{s_0\}) \not\subseteq G(\{s_0, s_1\})$). $G$ does not have any fixpoints!



## Fixpoints and ModelChecking

• Monotone functions *always* have a least and a greatest fixpoint.
• Meaning of EG, AF, EU can be expressed via greatest or least fixpoints of monotone functions on $\mathcal{P}(S)$.

Notation: $F^i(X)$ means that $F$ is applied $i$ many times

Ex: $F(Y) = Y \cup \{s_0\}$.
Then $F^2(Y) = F(F(Y)) = ...$

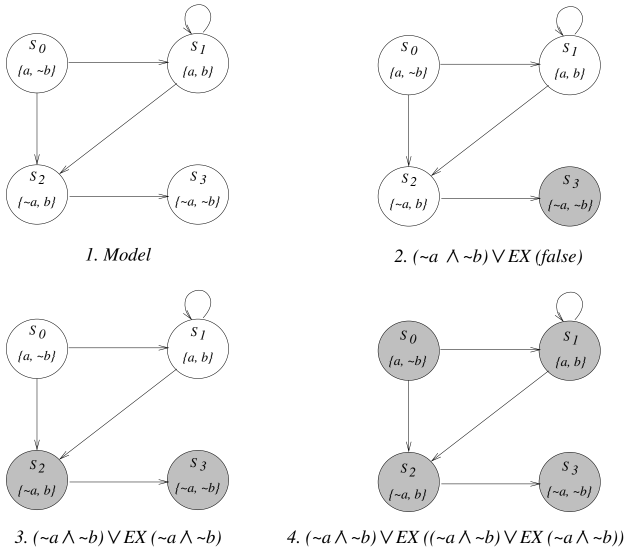
When will fixpoints be achieved? The Knaster-Tarski Theorem:
Let $S$ be a set $\{s_0, s_1, ..., s_b\}$ with $n + 1$ elements. If $F : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ is a monotone function, then $F^{n+1}(\emptyset)$ is the least fixpoint of $F$ and $F^{n+1}(S)$ is the greatest fixpoint of $F$. For proof, see section 3.9 of the book.

More notation: $\mu y.F(y)$ means the least fixpoint of $F(y)$; $\nu y.F(y)$ means the greatest fixpoint of $F(y)$.



## Example

Model checking $s_0 \models \mathrm{EF}(\neg a \wedge \neg b)$.




1. Model

2. (~a ∧ ~b) ∨ EX (false)

3. (~a ∧ ~b) ∨ EX (~a ∧ ~b)

4. (~a ∧ ~b) ∨ EX ((~a ∧ ~b) ∨ EX (~a ∧ ~b))

## Symbolic Model Checking

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

- Formulas represent sets of states that satisfy it

$a$ - set of states in which $a$ is true - $(\{s_0, s_1\})$

$b$ - set of states in which $b$ is true - $(\{s_1, s_2\})$

$a \vee b = \{s_0, s_1, s_2\}$

- State transitions are described by relations over two sets of variables, $V$ (source state) and $V\prime$ (destination state)

Transition from $s_2$ to $s_3$ is described by $(\neg a \wedge b \wedge \neg a\prime \wedge \neg b\prime)$.

Transition from $s_0$ to $s_1$ and $s_2$, and from $s_1$ to $s_2$ and to itself is described by $(a \wedge b\prime)$.

Relation $R$ is described by $(a \wedge b\prime) \vee (\neg a \wedge b \wedge \neg a\prime \wedge \neg b\prime)$



## Symbolic model checking (Cont'd)

The meaning for CTL formulas can be redefined in terms of sets of states:

$$
\begin{array}{rll}
s \models f & \text{iff} & s \in f \text{ where } f \in V \\
s \models \neg f & \text{iff} & s \in \neg f \\
s \models f \vee g & \text{iff} & s \in (f \vee g) \\
s \models \mathsf{EX} f & \text{iff} & s \in (\exists V'(R \wedge f(V/V'))) \\
s \models \mathsf{AX} f & \text{iff} & s \in \neg(\exists V'(R \wedge \neg f(V/V'))) \\
s \models \mathsf{E}(f \mathsf{U} g) & \text{iff} & s \in \mu y.(g \vee (f \wedge \mathsf{EX} \neg y)) \\
s \models \mathsf{A}(f \mathsf{U} g) & \text{iff} & s \in \mu y.(g \vee (f \wedge \mathsf{AX} y))
\end{array}
$$



## Symbolic model checking

- A CTL formula $f$ is evaluated for a model by deriving a *propositional logic expression* that describes the set of states satisfying the CTL formula for the model.
- The model checker verifies that the interpretation of the model's initial state $s_0$ satisfies the expression.

Example - check $s_0 \models \mathsf{EX}(\neg a \wedge \neg b)$, i.e., compute a formula representing the states that have successors where $(\neg a \wedge \neg b)$ is true:
- $R$ - the transition relation
- $f$ - the formula being checked
- $f(a, b/a\prime, b\prime)$ - substitution, leading to reasoning about the next state
- Replace formulas like $\exists v, (f)$ by $(f(v/true) \vee f(v/false)$.



## Computation

$$
\begin{array}{l}
EX(\neg a \wedge \neg b) = \\
\exists a', b'(((a \wedge b') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')) \\
\quad \wedge((\neg a \wedge \neg b)(a, b/a', b'))) = \\
\exists a', b'(((a \wedge b') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')) \\
\quad \wedge(\neg a' \wedge \neg b')) = \\
\exists a', b'((a \wedge b' \wedge \neg a' \wedge \neg b') \\
\quad \vee(\neg a \wedge b \wedge \neg a' \wedge \neg b' \wedge \neg a' \wedge \neg b')) = \\
\exists a', b'((false) \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')) = \\
\exists a', b'(\neg a \wedge b \wedge \neg a' \wedge \neg b') = \\
\exists a'((\neg a \wedge b \wedge \neg a' \wedge \neg true) \vee \\
\quad (\neg a \wedge b \wedge \neg a' \wedge \neg false)) = \\
\exists a'(\neg a \wedge b \wedge \neg a') = (\neg a \wedge b \wedge \neg true) \\
\quad \vee(\neg a \wedge b \wedge \neg false) = \neg a \wedge b
\end{array}
$$

## Evaluation of example

The computed propositional logic formula represents the set of states whose interpretations satisfy EX($\neg a \wedge \neg b$), that is, $\{s_2\}$.

EX($\neg a \wedge \neg b$) is a theorem (i.e., $s_0 \models$ EX($\neg a \wedge \neg b$)) if the values of $a$ and $b$ in $s_0$ satisfy $\neg a \wedge b$.

In $s_0$, $a$=true, $b$=false. So, $s_0 \not\models \neg a \wedge b$. Thus, $s_0 \not\models$ EX($\neg a \wedge \neg b$).



## Symbolic model checking

Example: calculate EF($\neg a \wedge \neg b$) for our model:

1) $(\neg a \wedge \neg b) \vee EX false = (\neg a \wedge \neg b)$
2) $(\neg a \wedge \neg b) \vee EX(\neg a \wedge \neg b) = (\neg a \wedge \neg b) \vee (\neg a \wedge b) = \neg a$
3) $(\neg a \wedge \neg b) \vee EX(\neg a) =$
$(\neg a \wedge \neg b) \vee \exists a', b'(((a \wedge b') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b'))$
$\wedge((\neg a)(a/a'))) =$
$(\neg a \wedge \neg b) \vee \exists a', b'(((a \wedge b') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')) \wedge \neg a') =$
$(\neg a \wedge \neg b) \vee \exists a', b'((a \wedge b' \wedge \neg a') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')) =$
$(\neg a \wedge \neg b) \vee \exists a'(((a \wedge true \wedge \neg a') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg true))$
$\vee((a \wedge false \wedge \neg a') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg false))) =$
$(\neg a \wedge \neg b) \vee \exists a'((a \wedge \neg a') \vee (\neg a \wedge b \wedge \neg a')) =$
$(\neg a \wedge \neg b) \vee ((a \wedge \neg true)$
$\vee(\neg a \wedge b \wedge \neg true)) \vee ((a \wedge \neg false) \vee (\neg a \wedge b \wedge \neg false)) =$
$(\neg a \wedge \neg b) \vee ((false) \vee (false)) \vee ((a) \vee (\neg a \wedge b)) =$
$(\neg a \wedge \neg b) \vee (\neg a \wedge b) \vee a = true$
4) $(\neg a \wedge \neg b) \vee EX(true) = true$



## Symbolic model checking

Note that the formulas at the end of each step correspond to the set of states that is shaded after that step.

- The first formula, ($\neg a \wedge \neg b$), corresponds to $\{s_3\}$
- The second formula, ($\neg a$), corresponds to $\{s_2, s_3\}$
- The last formula, *true*, corresponds to the set of all states.



## Pros and Cons

• Often cannot express full requirements
  – instead, check several smaller properties
• Few real systems have sufficiently small state space to allow direct checking
  – must generally abstract them or "downscale" them. Abstractions may enable checking systems with virtually unlimited number of states
• Largely automatic and fast
• Produces counterexamples
• Can handle systems with 100-200 state variables
• Generally used for debugging rather than assurance
• Usually, find more problems by exploring *all* the behaviors of a downscaled system than by testing only *some* of the behaviors of full system.

## Model-checking Examples (Up to 1996)

**IEEE Futurebus+** (1992, Clarke, CMU)

- Used SMV to verify the cache coherence protocol described in IEEE Futurebus+ Standard 896.1-1991
- Discovered a number of previously undetected errors in the design
- First attempt to find errors in an IEEE standard

**IEEE SCI** (1992, Dill, Stanford)

- Murphi verification system
- Verified cache coherence protocol of Scalable Coherent Interface, IEEE Standard 1596-1992
- Based their model directly on the C code given as definition of the standard
- Verified just small instances of the system
- Found several errors, ranging from omissions of variable initializations ti subtle logical errors
- The protocol has already been simulated and implemented!!



## Model-checking Examples (Cont'd)

**Stereo Components** (1994-96)

- 1994. Proved (manually) correctness of a control protocol used in Philips stereo components
- 1995. Verified an abstraction of protocol using symbolic model checker HyTech and inferred more efficient timing of the protocol
- 1996. Model checked the entire protocol

**ISDN/ISUP** (1989-92)

- NewCoRe Project - first full-scale application of formal verification methods in routine software design project within AT&T
- 5 people formalized 145 requirements in temporal logic and did proofs with a special-purpose model checker
- 7500 lines of Specification and Description Language (SDL) source code verified
- 112 errors were revealed in high-level designs



## Model-checking Examples (Cont'd)

**PowerScale** (1995, Bull with Berimag Laboratory)

- Used LOTOS to describe processors, memory controller, and bus arbiter of the multiprocessor architecture PowerScale
- Architecture is based on IBM's Power PC microprocessor
- Identified four correctness properties expressing essential requirements for proper functioning of the arbitration algorithm
- Established correctness automatically in a few minutes



## Model-checking Examples (Cont'd)

**Buildings** (1995, North Caroline State)

- Used Concurrency Workbench to analyze timing properties of a distributed active structural control system
- System was designed to make buildings more resistant to earthquakes
- First coded in a timed version of the CCS. Resulting model contained over $2.12 * 10^{19}$ states and was not analyzable
- Applied semantic minimization feature of Concurrency Workbench
- Analyzed properties and uncovered an error in a timer setting
- If undetected, this error would cause active structural control component to worsen, rather than dampen, the vibration experienced by buildings during earthquakes

## State of the Art Model Checkers

Temporal logic model checkers

- EMC - Clarke and Emerson 81-86
- CESAR - Queille and Sifakis 1982
- SMV - McMillan 1993 - first to use BDDs
- Spin - Gerth et al. 1995 - uses partial order reduction to reduce the state explosion problem
- Murphi - Dill et al. 1992 is based on Unity programming language
- Concurrency Workbench - Cleaveland et al. 1993 - verifies CCS processes for properties expressed as mu-calculus formulas
- HyTech - Alur et al. 1996 - for hybrid systems
- Kronos - Daws and Yovine 1995 - for real-time systems
- FORMAT - Damm et al. 1996, SVE - Filkorn et al. 1994 and CV - Deharbe and Borrione 1995 - focus on hardware verification



## State of the Art Model Checkers (Cont'd)

Behavior conformance checkers

- Cospan/FormalCheck system - DePalma and Glaser 1996 - based on showing inclusion between omega automata
- FDR - Roscoe 1994 - checks refinement between CSP programs
- Concurrency Workbench checks a similar notion of refinement between CCS programs
- Concurrency Workbench and Auto - Roy and de Simone 1990 - can be used to minimize systems with respect to observational equivalence and to determine if two systems are observably correct



## State of the Art Model Checkers (Cont'd)

Combination checkers

- Berkeley's HSIS - Hojati et al. 1993 - combines model checking with language inclusion
- Stanford's STep - Bjorner et al. 1996 - with deductive methods
- VIS - Vrayton et al. 1996 - with logic synthesis
- PVS Theorem prover - Owre, Rushby, Shankar 1992 - has a model checker for the model mu-calculus
- METAFrame - Steffen et al. 1996 - an environment that supports model checking in the entire software development process.



food for slide eater