Model-Checking

Readings: Textbook, 3.4-3.7, 3.9.

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

CTL – Review

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 \( \neg f, f \land g, f \lor g, AXf, EXf, A[fUg], E[fUg], AFf, EFf, AGf, EGf \).

Temporal operators - quantifier (A or E) followed by F (future), G (global), U (until), or X (next).
Where did the "tree" come from?

see picture

**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:

- $\text{EX}(f)$ (AX($f$)) is true in $M, s_i$ if $f$ is true in some (all) successor of $M, s_i$

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

\[ M, s \models p \iff (I(s))(p) \]
\[ M, s \models \neg f \iff s \not\models f \]
\[ M, s \models f \lor g \iff s \models f \lor s \models g \]
\[ M, s_0 \models \text{EX} \neg f \iff \exists \text{ path } (s_0, s_1, \ldots), s_1 \models \neg f \]
\[ M, s_0 \models \text{AX} \neg f \iff \forall \text{ paths } (s_0, s_1, \ldots), s_1 \models \neg f \]
\[ M, s_0 \models \text{E}(f \lor g) \iff \exists \text{ path } (s_0, s_1, \ldots), \exists \text{ i.t. } M, s_i \models g \text{ and } \forall j < i, s_j \models f \]
\[ M, s_0 \models \text{A}(f \lor g) \iff \forall \text{ paths } (s_0, s_1, \ldots), \exists \text{ i.t. } M, s_i \models g \text{ and } \forall j < i, s_j \models f \]

**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. \( f_0 = \text{TRANSLATE}(f_0) \) (in terms of AF, EU, EX, \( \land, \lor, \bot \))

2. Label the states of \( M \) with the subformulas of \( f_0 \) that are satisfied there and working outwards towards \( f_0 \).

Ex: \( \text{AF}(a \land \text{E}(b \lor c)) \)

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:

- $\bot$: then no states are labeled with $\bot$.
- $p$ (prop. formula): label $s$ with $p$ if $p \in I(s)$.
- $\psi_1 \land \psi_2$: label $s$ with $\psi_1 \land \psi_2$ 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 $\psi_1$.
- $\mathit{AF} \; \psi_1$:
  - If any state $s$ is labeled with $\psi_1$, label it with $\mathit{AF} \; \psi_1$.
  - Repeat: label any state with $\mathit{AF} \; \psi_1$ if all successor states are labeled with $\mathit{AF} \; \psi_1$, until there is no change.

**Labeling Algorithm (Cont’d)**

- $\mathit{E} \; [\psi_1 \cup \psi_2]$:
  - If any state $s$ is labeled with $\psi_2$, label it with $\mathit{E} \; [\psi_1 \cup \psi_2]$.
  - Repeat: label any state with $\mathit{E} \; [\psi_1 \cup \psi_2]$ if it is labeled with $\psi_1$ and at least one of its successors is labeled with $\mathit{E} \; [\psi_1 \cup \psi_2]$, until there is no change.
- $\mathit{EX} \; \psi_1$: label any state with $\mathit{EX} \; \psi_1$ 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 $M, s_0 \models E(a \cup \neg b)$

1. Model
   - $s_0 \overset{a, b}{\rightarrow} s_2 \overset{a}{\rightarrow} s_3
     \overset{b}{\rightarrow} s_1$
2. mark with $\neg b$
   - $s_0 \overset{a, b}{\rightarrow} s_2 \overset{a}{\rightarrow} s_3
     \overset{b}{\rightarrow} s_1
3. label with $E[a \cup \neg b]$
   - $s_0 \overset{a, b}{\rightarrow} s_2 \overset{a}{\rightarrow} s_3
     \overset{b}{\rightarrow} s_1$
4. Continue labeling
   - $s_0 \overset{a, b}{\rightarrow} s_2 \overset{a}{\rightarrow} s_3
     \overset{b}{\rightarrow} s_1$

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.

$$\models_{EG}$$

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.
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.
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 subrange.

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
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;
  esac;
  next(semaphore) := case
    state = entering : 1;
    state = exiting : 0;
  esac;
FAIRNESS
running

Is the Spec Correct?

-- specification AG (!(proc1.state = critical &
  proc2.state) 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.state 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:

```plaintext
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 $\psi_1 \land \psi_2 \land \ldots \land \psi_n$ and $n$ fairness constraints $\psi_1, \psi_2, \ldots, \psi_n$, written on separate lines under FAIRNESS? Write an SMV program with a fairness constraint $a \land b$ 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!