Model Checking

• Two Lectures
  - Model Checking
  - Software Model Checking
  - SLAM and BLAST

• “Flying Boxes”
  - It is traditional to describe this stuff (especially SLAM and BLAST) with high-gloss animation. Sorry.

• Some Key Players:
  - Model Checking: Ed Clarke, Ken McMillan, Amir Pnueli
  - SLAM: Tom Ball, Sriram Rajamani
  - BLAST: Ranjit Jhala, Rupak Majumdar, Tom Henzinger

Take-Home Message

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

Overarching Plan

• [Model Checking] (Today)
  - Transition Systems (Models)
  - Temporal Properties
  - LTL and CTL
  - (Explicit State) Model Checking
  - Symbolic Model Checking

• Counterexample Guided Abstraction Refinement
  - Safety Properties
  - Predicate Abstraction (“c2bp”)
  - Software Model Checking (“bebop”)
  - Counterexample Feasibility (“newton”, “hw 5”)
  - Abstraction Refinement (weakest pre, thrm prvr)

Spoiler Space

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

Topic:

( Generic) Model Checking

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

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

Verification vs. Falsification

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

Temporal Properties

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

An Example Concurrent Program

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>while True do</td>
</tr>
<tr>
<td>11:</td>
<td>wait(turn = 0); // critical section</td>
</tr>
<tr>
<td>12:</td>
<td>turn := 1;</td>
</tr>
<tr>
<td>13:</td>
<td>end while;</td>
</tr>
<tr>
<td>20:</td>
<td>while True do</td>
</tr>
<tr>
<td>21:</td>
<td>wait(turn = 1); // critical section</td>
</tr>
<tr>
<td>22:</td>
<td>turn := 0;</td>
</tr>
<tr>
<td>23:</td>
<td>end while</td>
</tr>
</tbody>
</table>

Reachable States of the Example Program

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

Next: formalize this intuition ...

Transition Systems

- In model checking the system being analyzed is represented as a labeled transition system $T = (S, I, R, L)$
  - Also called a Kripke Structure
  - $S$ = Set of states // standard FSM
  - $I \subseteq S$ = Set of initial states // standard FSM
  - $R \subseteq S \times S$ = Transition relation // standard FSM
  - $\Lambda: S \rightarrow \mathcal{P}(AP)$ = Labeling function // this is new!
- AP: Set of atomic propositions (e.g., “x=5”)
  - Atomic propositions capture basic properties
  - For software, atomic props depend on variable values
  - The labeling function labels each state with the set of propositions true in that state
Properties of the Program

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

Temporal Logics

- There are four basic temporal operators:
  1) $X p = \text{Next } p$, $p$ holds in the next state
  2) $G p = \text{Globally } p$, $p$ holds in every state, $p$ is an invariant
  3) $F p = \text{Future } p$, $p$ will hold in a future state, $p$ holds eventually
  4) $p U q = \text{Until } q$, assertion $p$ will hold until $q$ holds
- Precise meaning of these temporal operators are defined on execution paths

Execution Paths

- A path in a transition system is an infinite sequence of states
  - $(s_0, s_1, s_2, ...)$, such that $\forall i \geq 0. (s_i, s_{i+1}) \in R$
- A path $(s_0, s_1, s_2, ...)$ is an execution path if $s_0 \in I$
- Given a path $x = (s_0, s_1, s_2, ...)$
  - $x_i$ denotes the $i$th state $s_i$
  - $x_i$ denotes the $i$th suffix $(s_i, s_{i+1}, s_{i+2}, ...)$
- In some temporal logics one can quantify the paths starting from a state using path quantifiers
  - $A$: for all paths
  - $E$: there exists a path

Linear Time Logic (LTL)

- LTL properties are constructed from atomic propositions in AP; logical operators $\land, \lor, \neg$; and temporal operators $X, G, F, U$.
- The semantics of LTL properties is defined on paths:
  Given a path $x$:
  - $x \models p$ iff $L(x, p)$ // atomic prop
  - $x \models X p$ iff $x_1 \models p$ // next
  - $x \models F p$ iff $\exists i \geq 0. x_i \models p$ // future
  - $x \models G p$ iff $\forall i \geq 0. x_i \models p$ // globally
  - $x \models p U q$ iff $\exists i \geq 0. x_i \models q$ and $\forall j < i. x_j \models p$ // until

Satisfying Linear Time Logic

- Given a transition system $T = (S, I, R, L)$ and an LTL property $p$, $T$ satisfies $p$ if all paths starting from all initial states $I$ satisfy $p$
- Examples:
  - $\text{Invariant}(\neg((pc1=12 \land pc2=22)))$
  - $\text{Eventually}(pc1=12)$

Computation Tree Logic (CTL)

- In CTL temporal properties use path quantifiers
  - $A$: for all paths
  - $E$: there exists a path
- The semantics of CTL properties is defined on states:
  Given a path $x$
  - $s \models p$ iff $L(s, p)$
  - $s \models \exists x p$ iff $\exists$ a path $(s_0, s_1, s_2, ...). s_i \models p$
  - $s \models AX p$ iff $\forall$ paths $(s_0, s_1, s_2, ...). s_i \models p$
  - $s \models EG p$ iff $\exists$ a path $(s_0, s_1, s_2, ...), \forall i \geq 0. s_i \models p$
  - $s \models AG p$ iff $\forall$ paths $(s_0, s_1, s_2, ...). \forall i \geq 0. s_i \models p$
Linear vs. Branching Time

- **LTL** is a **linear time logic**
  - When determining if a path satisfies an LTL formula we are only concerned with a **single path**

- **CTL** is a **branching time logic**
  - When determining if a state satisfies a CTL formula we are concerned with multiple paths
  - In CTL the computation is not viewed as a single path but as a **computation tree** which contains all the paths
  - The computation tree is obtained by unrolling the transition relation

- The expressive powers of CTL and LTL are **incomparable**
  - Basic temporal properties can be expressed in both logics
  - Not in this lecture, sorry! (Take a class on Modal Logics)

Remember the Example

One path starting at state

(turn=0, pc1=10, pc2=20)

A computation tree starting at state

(turn=0, pc1=10, pc2=20)

LTL Satisfiability Examples

- $\Box p$ does not hold
- $p$ holds

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

On this path: $F p$ holds, $G p$ holds, $p$ holds, $X p$ holds, $X (X p)$ holds, $X (X (X p))$ holds

CTL Examples

- $\Box p$ does not hold
- $p$ holds

At state $s$:
- $EF p$, $EX (EX p)$, $AF (-p)$, $\neg p$ holds
- $AF p$, $AG p$, $AG (-p)$, $EX p$, $EG p$, $p$ does not hold

At state $s$:
- $EF p$, $AF p$, $EX (EX p)$, $EX p$, $EG p$, $p$ holds
- $AG p$, $AG (-p)$, $AF (-p)$ does not hold

At state $s$:
- $EF p$, $AF p$, $AG p$, $EG p$, $p$ holds
- $EX p$, $AX p$, $AX p$, $p$ holds
- $EG (-p)$, $EF (-p)$, $\neg p$ does not hold

Model Checking Complexity

- Given a transition system $T = (S, I, R, L)$ and a CTL formula $f$
  - One can check if a state of the transition system satisfies the temporal logic formula $f$ in $O(|f| \times (|S| + |R|))$ time

- Given a transition system $T = (S, I, R, L)$ and an LTL formula $f$
  - One can check if the transition system satisfies the temporal logic formula $f$ in $O(2^{|f|} \times (|S| + |R|))$ time

- Model checking procedures can **generate counter-examples without increasing the complexity of verification (= “for free”)**
State Space Explosion

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

Explicit-State Model Checking

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

Temporal Properties ≡ Fixpoints

- States that satisfy \(AG(p)\) are all the states which are not in \(EF(\neg p)\) (≠ the states that can reach \(\neg p\))
- Compute \(EF(\neg p)\) as the fixpoint of \(Func: 2^S \rightarrow 2^S\)
  - Given \(Z \subseteq S\),
    - \(Func(Z) = \neg p \cup \text{reach-in-one-step}(Z)\)
    - or \(Func(Z) = \neg p \cup EX(Z)\)
- Actually, \(EF(\neg p)\) is the least-fixpoint of \(Func\)
  - smallest set \(Z\) such that \(Z = Func(Z)\)
  - to compute the least fixpoint, start the iteration from \(Z=\emptyset\), and apply the \(Func\) until you reach a fixpoint
  - This can be computed (unlike most other fixpoints)

Pictorial Backward Fixpoint

- Initial states that violate \(AG(p)\) = initial states that satisfy \(EF(\neg p)\)
  - Initial states that violate \(AG(p)\) = initial states that can reach \(\neg p\)
  - States that can reach \(\neg p\) = \(EF(\neg p)\)
  - States that violate \(AG(p)\) = \(EX(\neg p)\)

This fixpoint computation can be used for:
  - verification of \(EF(\neg p)\)
  - or falsification of \(AG(p)\)
  - ... and a similar forward fixpoint handles the rest

Symbolic Model Checking

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

Binary Decision Diagrams (BDDs)

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

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

Where’s the Beef

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

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

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

Key Terms

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

So ... what is Counterexample Guided Abstraction Refinement?
- Theorem Proving?
- Dataflow Analysis?
- Model Checking?

Verification by Theorem Proving

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

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

- Loop Invariants
- Multithreaded Programs
+ Behaviors encoded in logic
+ Decision Procedures

Example
```c
1: do{
2: lock();
3: old = new;
4: q = q->next;
5: if (q != NULL){
6: q->data = new;
7: unlock();
8: new ++;
9: }
10: } while(new != old);
11: unlock();
12: return;
}
```

-- Verification by Program Analysis

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

- Imprecision due to fixed facts
+ Abstraction
+ Type/Flow Analyses

Example
```c
1: do{
2: lock();
3: old = new;
4: q = q->next;
5: if (q != NULL){
6: q->data = new;
7: unlock();
8: new ++;
9: }
10: } while(new != old);
11: unlock();
12: return;
}
```

-- Verification by Model Checking

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

- Pgm → Finite state model
- State explosion
+ State Exploration
+ Counterexamples

Example
```c
1: do{
2: lock();
3: old = new;
4: q = q->next;
5: if (q != NULL){
6: q->data = new;
7: unlock();
8: new ++;
9: }
10: } while(new != old);
11: unlock();
12: return;
}
```

-- Combining Strengths

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

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

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

Homework

- Project Due!
  - Need help? Stop by my office or send email.