Model Checking: An Introduction

Meeting 2

Fundamentals of Programming Languages
CSCI 5535, Spring 2009

Announcements

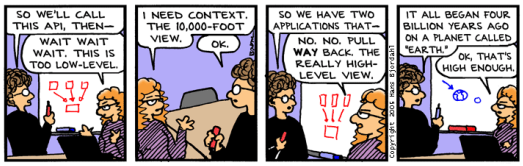
- Office hours
  - M 1:30pm-2:30pm
  - W 5:30pm-6:30pm (after class)
  - and by appointment
  - ECOT 621
- Moodle problems?

A Double Header

- Two lectures
  - Model checking primer
  - Software model checking
  - SLAM and BLAST (tools)
- Some key players:
  - Model checking
    Ed Clarke, Ken McMillan, Amir Pnueli
  - SLAM
    Tom Ball, Sriram Rajamani
  - BLAST
    Ranjit Jhala, Rupak Majumdar, Tom Henzinger

Who are we again?

- We're going to find critical bugs in important bits of software
  - using PL techniques!
- You'll be enthusiastic about this
  - and thus want to learn the gritty details



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 **counterexamples**.
- 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 (i.e., models)
- Temporal properties
- Temporal logics: 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"
- Abstraction refinement — weakest pre, thrm prv

Spoiler

- This stuff really works!
- Symbolic model checking is a massive success in the model-checking field
- SLAM took the PL world by storm
  - Spawned multiple copycat projects
  - Launched Microsoft's Static Driver Verifier (released in the Windows DDK)

7

Model Checking

There are complete courses in model checking (see ECEN 5139, Prof. Somenzi).

Model Checking by Edmund M. Clarke, Orna Grumberg, and Doron A. Peled.
Symbolic Model Checking by Ken McMillan.

We will skim.

Keywords

Model checking is an automated technique
Model checking verifies transition systems
Model checking verifies temporal properties
Model checking falsifies by generating counterexamples

A model checker is a program that checks if a (transition) system satisfies a (temporal) property

9

Verification vs. Falsification

- What is verification?
  - prove that a property of a system holds
- What is falsification?
  - disprove that a property holds

10

Verification vs. Falsification

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

11

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

G, AG, □ ("globally" 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, ◇ ("future" or "diamond" or "exists")

12

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

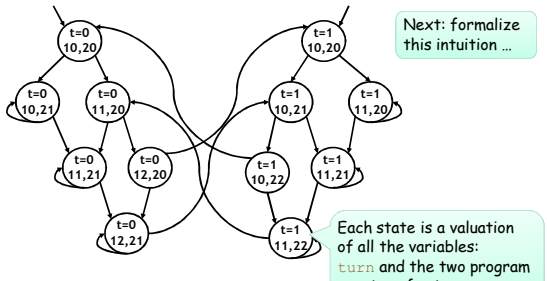
```
10: while (true) {
11:   wait(turn == 0);
      // critical section
12:   work(); turn = 1;
13: }

|| // concurrently with

20: while (true) {
21:   wait(turn == 1);
      // critical section
22:   work(); turn = 0;
23: }
```

13

Reachable States of the Example Program



Next: formalize this intuition …

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

Analyzed System is a Transition System

- 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
  - $L: S \to 2^{AP}$ = Labeling function // this is new!
- AP: Set of atomic propositions (e.g., "x=5"$\in$AP)
  - 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

15

Example Properties of the Program

- "In all the reachable states (configurations) of the system, the two processes are *never in the critical section at the same time*"
  - "pc1=12", "pc2=22" are atomic properties for being in the critical section

Invariant(¬(pc1=12 ∧ pc2=22))

- "*Eventually the first process enters* the critical section"

Eventually(pc1=12)

16

Example Properties of the Program

- "In all the reachable states (configurations) of the system, the two processes are *never in the critical section at the same time*"
  - "pc1=12", "pc2=22" are atomic properties for being in the critical section

Invariant(¬(**pc1=12 ∧ pc2=22**))

- "*Eventually the first process enters* the critical section"

Eventually(**pc1=12**)

17

Temporal Logics

For what?
For expressing properties

There are four basic temporal operators:

- X p
  Next p, p holds in the next state
- G p
  Globally p, p holds in every state, p is an invariant
- F p
  Future p, p will hold in a future state, p holds eventually
- p U q
  p Until q, assertion p will hold until q holds
- Precise meaning of these temporal operators are defined on execution paths

18

Execution Paths

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

19

Paths and Predicates

- We write

  $x \models p$

  "the path x makes the predicate p true"
  - x is a path in a transition system
  - p is a temporal logic predicate
- Example:

  $A\ x.\quad x \models G\ (\neg(pc1=12 \wedge pc2=22))$

20

Linear Time Logic (LTL)

$T=(S, I, R, L)$

- LTL properties are constructed from atomic propositions in AP; logical operators $\wedge, \vee, \neg$; and temporal operators X, G, F, U.
- The semantics of LTL is defined on paths:

Given a path x: (e.g. $x = \tau$)

$x \models p$	iff	$L(x_0, p)$	*atomic prop*
$x \models X\ p$	iff	$x^1 \models p$	*next*
$x \models F\ p$	iff	$\exists i \geq 0.\ x^i \models p$	(FG ...)
$x \models G\ p$	iff	$\forall i \geq 0.\ x^i \models p$	
$x \models p\ U\ q$	iff	$\exists i \geq 0.\ \forall j < i.\ x^i \models q \wedge x^j \models p$	

path, LTL formula

21

Linear Time Logic (LTL)

- LTL properties are constructed from atomic propositions in AP; logical operators $\wedge, \vee, \neg$; and temporal operators X, G, F, U.
- The semantics of LTL is defined on paths:

Given a path x:

$x \models p$	iff	$L(x_0, p)$	*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*

22

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

23

Computation Tree Logic (CTL)

$T=(S, I, R, L)$

- In CTL temporal properties use path quantifiers: A : for all paths, E : there exists a path
- The semantics of CTL is defined on states:

Given a state s

$s \models p$	iff	$L(s, p)$
$s_0 \models EX\ p$	iff	$\exists$ a path $(s_0, s_1, s_2, \ldots).\ s_1 \models p$
$s_0 \models AX\ p$	iff	$\forall$ paths $(s_0, s_1, s_2, \ldots).\ s_1 \models p$
$s_0 \models EG\ p$	iff	$\exists$ a path $(s_0, s_1, s_2, \ldots).\ \forall i \geq 0.\ s_i \models p$
$s_0 \models AG\ p$	iff	$\forall$ paths $(s_0, s_1, s_2, \ldots).\ \forall i \geq 0.\ s_i \models p$

: state, CTL formula

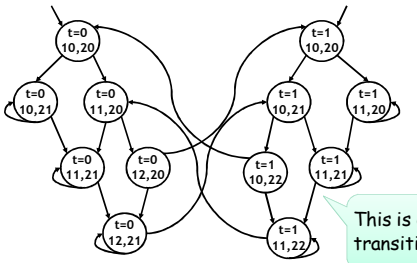
24

Linear vs. Branching Time

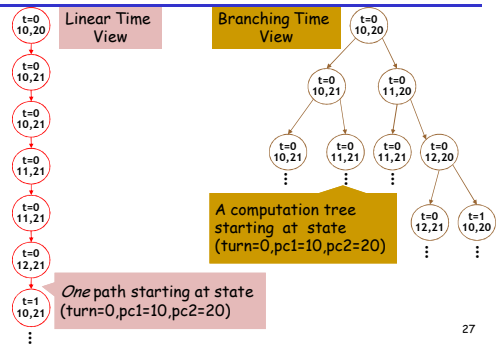
- LTL is a linear time logic
  - When determining if a path satisfies an LTL formula we are only concerned with a single path
- CTL is a branching time logic
  - When determining if a state satisfies a CTL formula we are concerned with multiple paths
  - In CTL the computation is instead viewed as a 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 (LTL $\subseteq$ CTL*, CTL $\subseteq$ CTL*)
- Basic temporal properties can be expressed in both logics
- Not in this lecture, sorry! (Take a class on Modal Logics)

Recall the Example

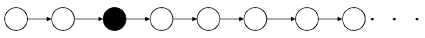


Linear vs. Branching Time



LTL Satisfiability Examples

○ p does not hold ● p holds



On this path:

Holds: $\neg p$, Fp, $X(Xp)$

Does Not Hold: Gp

LTL Satisfiability Examples

○ p does not hold ● p holds

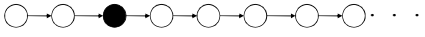
On this path:

Holds: Xp, p, Fp, Gp, $X(Xp)$

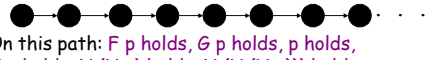
Does Not Hold

LTL Satisfiability Examples

○ p does not hold ● p holds



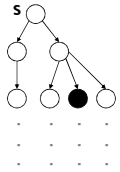
On this path: F p holds, G p does not hold, p does not hold, X p does not hold, X (X p) holds, X (X (X p)) does not hold



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

CTL Satisfiability Examples

○ p does not hold
● p holds



At state s:

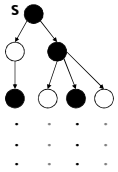
Holds: EF p, EX(EX p), AX(EX ¬p)

Does Not Hold

AF, AG, ER, EG, Ex, AX

CTL Satisfiability Examples

○ p does not hold
● p holds



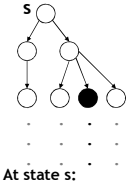
At state s:

Holds: EX(EX p), EX p, AX(EX p), EG p, AF p, EF p, p, EF ¬p

Does Not Hold

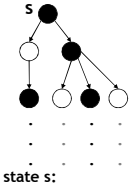
CTL Satisfiability Examples

○ p does not hold
● p holds



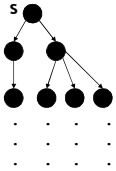
At state s:
EF p, EX (EX p),
AF (¬p), ¬p holds

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,
Ex p, AX p, p holds

EG (¬ p), EF (¬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 formula f in $O(|f| \times (|S| + |R|))$ time
  - Multiple depth first searches (one for each temporal operator) = explicit-state model checking
- Given a transition system T = (S, I, R, L) and an LTL formula f
  - One can check if the transition system satisfies the formula f in $O(2^{|f|} \times (|S| + |R|))$ time

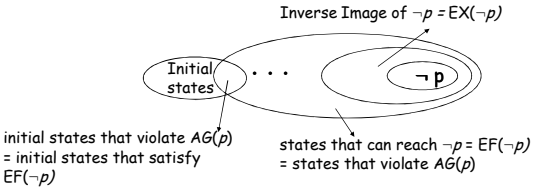
State Space Explosion

- 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

Algorithm: Temporal Properties = Fixpoints

- States that satisfy AG(p) are all the states which are not in EF(¬p) (= the states that can reach ¬p)
- Compute EF(¬p) as the **fixed point** of Func: $2^S \rightarrow 2^S$
- Given $Z \subseteq S$,
  - Func(Z) = ¬p ∪ **reach-in-one-step**(Z)
  - or Func(Z) = ¬p ∪ EX(Z)
- Actually, EF(¬p) is the ***least*-fixed point** of Func
  - smallest set Z such that Z = Func(Z)
  - to compute the least fixed point, start the iteration from Z=∅, and apply the Func until you reach a fixed point
  - This can be computed (unlike most other fixed points)

Pictorial Backward Fixed Point



This fixed point computation can be used for:

- verification of EF(¬p)
- or falsification of AG(p)

... and similar fixed points handle the other cases

Symbolic Model Checking

- Symbolic model checking represent state sets and the transition relation as Boolean logic formulas
  - Fixed point computations manipulate sets of states rather than individual states
  - Recall: we needed to compute EX(Z), but $Z \subseteq S$
- Fixed points can be computed by iteratively manipulating these formulas
- Use an efficient data structure for manipulation of Boolean logic formulas
  - Binary Decision Diagrams (BDDs)
- SMV (Symbolic Model Verifier) was the first CTL model checker to use 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

Building Up To Software Model Checking via Counterexample Guided Abstraction Refinement

There are easily dozens of papers.

We will skim.

Key Terms

- Counterexample guided abstraction refinement (CEGAR)
  - A successful software model-checking approach. Sometimes called "Iterative Abstraction Refinement".
- SLAM = The first CEGAR project/tool.
  - Developed at MSR
- Lazy Abstraction = CEGAR optimization
  - Used in the BLAST tool from Berkeley.

What *is* Counterexample Guided Abstraction Refinement (CEGAR)?

Verification by …

Model Checking?

Theorem Proving?

Dataflow Analysis or Program Analysis?

43

Verification

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

Is this program correct?

What does correct mean?

How do we determine if a program is correct?

44

Verification by Model Checking

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

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

- Program→Finite state model
- State explosion
+ State exploration
+ Counterexamples

Precise [SPIN,SMV,Bandera,JPF]

45

Verification by Theorem Proving

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

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

Invariant:

lock ∧ new = old

∨

¬lock ∧ new ≠ old

46

Verification by Theorem Proving

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

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

- Loop invariants
- Multithreaded programs
+ Behaviors encoded in logic
+ Decision procedures

Precise [ESC,PCC]

47

Verification by Program Analysis

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

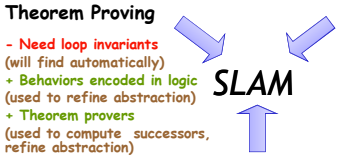
1. Dataflow Facts
2. Constraint System
3. Solve Constraints

- Imprecision: fixed facts
+ Abstraction
+ Type/Flow analyses

Scalable [Cqual,ESP]

48

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
(will find small good model)
+ State space exploration
(used to get a path sensitive analysis)
+ Counterexamples
(used to find relevant facts, refine abstraction)

49

For Next Time

- Read "Lazy Abstraction"
  - for the main ideas, ok to skim Sec. 7

50