CS 257: Introduction to Automated Reasoning

Model Checking, Bounded Model Checking, K-Induction, Interpolation
Outline

• What is Model Checking?
  • Modeling: Transition Systems
  • Specification: Linear Temporal Logic

• Historical Verification Approaches
  • Explicit-state
  • BDDs

• SAT/SMT-based Verification Approaches
  • Bounded Model Checking
  • K-Induction

• Inductive Invariants

* Many of the slides today are contributed by Makai Mann.
What is Model Checking?

- Approach for verifying the temporal behavior of a system
- **Model**: Representation of the system
- **Specification**: High-level desired property of system
- Considers infinite sequences

Model Checking Diagram:
- **Model**
- **Spec**
  - **Counter-Example**
  - **Proof** (optional)
Modeling: Transition System

• Model checking typically operates over *Transition Systems*
  • A (symbolic) state machine

• A Transition System is \( \langle S, I, T \rangle \)
  • \( S \): a set of states
  • \( I \): a set of initial states (sometimes use *Init* instead of \( I \) for clarity)
  • \( T \): a transition relation: \( T \subseteq S \times S \)
    • \( T(s_0, s_1) \) holds when there is a transition from \( s_0 \) to \( s_1 \)
Symbolic Transition Systems in Practice

• States are made up of state variables \( v \in V \)
  • A state is an assignment to all variables

• A Transition System is \( \langle V, I, T \rangle \)
  • \( V \): a set of state variables, \( V' \) denotes next state variables
  • \( I \): a set of initial states
  • \( T \): a transition relation
    • \( T(v_0, ..., v_n, v'_0, ..., v'_n) \) holds when there is a transition
    • Note: will often still use \( s \) to denote symbolic states (just know they’re made up of variables)

• Symbolic state machine is built by translating another representation
  • E.g. a program, a mathematical model, a hardware description, etc...
Symbolic Transition System Example

• 2 variables: \( V = \{v_0, v_1\} \)
  • \( S_0 := \neg v_0 \land \neg v_1, \quad S_1 := \neg v_0 \land v_1 \)
  • \( S_2 := v_0 \land \neg v_1, \quad S_3 := v_0 \land v_1 \)
• Transition relation
  \[
  \neg v_0 \land \neg v_1 \Rightarrow \left((\neg v'_0 \land v'_1) \lor (v'_0 \land \neg v'_1)\right) \land \\
  \neg v_0 \land v_1 \Rightarrow (v'_0 \land v'_1) \land \\
  (v_0 \land \neg v_1) \Rightarrow (v'_0 \land v'_1) \land \\
  (v_0 \land v_1) \Rightarrow (v'_0 \land v'_1)
  \]

S0 \rightarrow S1 \rightarrow S3
S0 \rightarrow S2
S2 \rightarrow S1
S1 \rightarrow S3
S3 \rightarrow S0
Modeling: Transition System Executions

• An execution is a sequence of states that respects \( I \) in the first state and \( T \) between every adjacent pair

• \( \pi \ := s_0 \ s_1 \ \ldots \ s_n \) is a finite sequence if \( I(s_0) \land \land_{i=1}^{n} T(s_{i-1}, s_i) \)
Meta Note: State Machine vs Execution Diagrams

State Machine uses capitals

Symbolic execution uses lowercase

Concrete Execution:
s0=S0, s1=S2, s2=S3, s3=S3
Specification: Linear Temporal Logic (LTL)

• Notation: \( M \models f \)
  • Transition system model, \( M \), entails LTL property, \( f \), for ALL possible paths
  • i.e. LTL is implicitly universally quantified

• Other logics include
  • CTL: computational tree logic (branching time)
  • CTL*: combination of LTL and CTL
  • MTL: metric temporal logic (for regions of time)
Specification: Linear Temporal Logic (LTL)

• Atomic state property \( P \subseteq S \):
  • Holds iff \( s_0 \in P \)

• **Next** \( P \): \( X(P) \)
  • \( P \) holds Next time
  • Also written \( \text{o}_p \)
  • True iff the next state meets property \( P \)

• **Invariant** \( P \): \( G(P) \)
  • \( P \) Globally holds
  • Also written \( \Box P \)
  • True iff every reachable state meets property \( P \)
Specification: Linear Temporal Logic

- **Eventually** $P$: $F(P)$
  - $P$ holds in the **Future**
  - Also written $\Diamond P$
  - True iff $P$ eventually holds

- **$P_1$ Until $P_2$:** $P_1 U P_2$
  - $P_1$ holds until $P_2$ holds
  - True iff $P_1$ holds up until (but not necessarily including) a state where $P_2$ holds
  - $P_2$ must hold at some point
Specification: Linear Temporal Logic

• LTL operators can be composed
  • $G(Req \Rightarrow F(Ack))$
    • Every request eventually acknowledged
  • $G(F(DeviceEnabled))$
    • The device is enabled infinitely often (from every state, it’s eventually enabled again)
  • $F(G(\neg Initializing))$
    • Eventually it’s not initializing
    • E.g. there is some initialization procedure that eventually ends and never restarts
Specification: Safety vs. Liveness

• Safety: “something bad does not happen”
  • State invariant, e.g. $G(\neg \text{bad})$

• Liveness: “something good eventually happens”
  • Eventuality, e.g. $GF(\text{good})$

• Fairness conditions
  • Fair traces satisfy each of the fairness conditions infinitely often
  • E.g. only fair if it doesn’t delay acknowledging a request forever

• Every property can be written as a conjunction of a safety and liveness property

Specification: Liveness to Safety

- Can reduce liveness to safety checking
- For SAT-based:
- Several approaches for first-order logic
- From now on, we consider only safety properties
Historical Verification Approaches: Explicit State

• Tableaux-style state exploration
• Form of depth-first search
• Many clever tricks for reducing search space
• Big contribution is handling temporal logics (including branching time)
Historical Verification Approaches: BDDs

- Binary Decision Diagrams (BDDs)
  - Manipulate sets of states symbolically


Historical Verification Approaches: BDDs

• Represent Boolean formula as a decision diagram
• Example: \((x_1 \land x_2) \lor (x_3 \land x_4)\)
• Can be much more succinct than other representations

Credit for Example: Introduction to Formal Hardware Verification – Thomas Kropf
Historical Verification Approaches: BDDs
BDD Operators

- Negation
  - Swap leaves (F → T)
- AND
  - All Boolean operators implemented recursively
- These two operators are sufficient

Image Credit: Introduction to Formal Hardware Verification – Thomas Kropf
BDDs: Cofactoring

• \( f\mid_{x_2} \) for BDD \( f \) is fixing \( x_2 \) to be negative

Credit for Example: Introduction to Formal Hardware Verification – Thomas Kropf
BDD Image Computation

- Current reachable states are BDD $R$
  - Over variable set $V$
- Compute next states with:
  - $N := \exists V T(V, V') \land R(V)$
  - Existential is implemented cofactoring: $\exists x_i . f(\ldots, x_i, \ldots) := f(\ldots, F, \ldots) \lor f(\ldots, T, \ldots)$
- Grow reachable states
  - $R = R \lor N[V'/V]$
  - Map next-state variables to current state, then add to reachable states

BDD image computation is based on the idea that all reachable next states are either already in $R$ or they are the result of applying the transition function to some set of states $V$ in $R$ to reach the set of states $V'$.
BDD-based model checking

• Start with $R = Init$

• Keep computing image and growing reachable states

• Stop when there’s a fixpoint (reachable states not growing)

• Can handle $\sim 10^{20}$ states
  • More with abstraction techniques and compositional model checking
BDD: Variable Ordering

• Good variable orderings can be exponentially more compact
  • Finding a good ordering is NP-complete

• There are formulas that have no non-exponential ordering
BDD for the function $f(x_1, \ldots, x_8) = x_1 x_2 + x_3 x_4 + x_5 x_6 + x_7 x_8$ using bad variable ordering

Good variable ordering

Image Credit: https://en.wikipedia.org/wiki/Binary_decision_diagram
SAT-based model checking

- Edmund Clarke
  - One of the founders of model checking
- SAT solving taking off
- Clarke hired several post-doctoral students to try to use SAT as an oracle to solve model checking problems
- Struggled for a while to find a general technique
  - What if you give up completeness? → Bounded Model Checking

Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu. Symbolic Model Checking without BDDs. TACAS 1999
Bounded Model Checking (BMC)

• Sacrifice completeness for quick bug-finding
• Unroll the transition system
  • Each variable $v \in V$ gets a new symbol for each time-step, e.g. $v_k$ is $v$ at time $k$
  • Space-Time duality: unrolls temporal behavior into space
• For increasing values of $k$, check:
  • $I(s_0) \land \bigwedge_{i=1}^k T(s_{i-1}, s_i) \land \neg P(s_k)$
• If it is ever SAT, return FALSE
  • Can construct a counter-example trace
BMC Graphically

$I(s_0)$

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$

$s_0$ must be an initial state

$P(s_k)?$ Check if it can violate the property at time $k$
Bounded Model Checking: Completeness

- Completeness condition: reaching the diameter
  - Diameter: $d$
    - Depth needed to unroll to such that every possible state is reachable in $d$ steps or less
    \[
    rd(M) := \min\{i | \forall s_0, \ldots, s_{i+1}. \exists s'_0, \ldots, s'_i.
    
    I(s_0) \land \wedge_{j=0}^{i-1} T(s_j, s_{j+1}) \rightarrow (I(s'_0) \land \wedge_{j=0}^{i-1} T(s'_j, s'_{j+1}) \land \forall j=0 \rightarrow s'_j = s_{i+1})\}
    \]
  - Recurrence diameter: $d_r$
    - The depth such that every execution of the system of length $\geq d_r$ must revisit states
    - Can be exponentially larger than the diameter
    \[
    rdr(M) := \max\{i | \exists s_0 \ldots s_i. I(s_0) \land \wedge_{j=0}^{i-1} T(s_j, s_{j+1}) \land \wedge_{j=0,k=j+1}^{i} s_j \neq s_k\}\]
    - $d_r \geq d$

- Very difficult to compute the diameter
  - Requires a quantifier: find $d$ such that any state reachable at $d + 1$ is also reachable in $\leq d$ steps (replace “$i$” with “$d$” in equation (3) above)
K-Induction

• Extends bounded model checking to be able to prove properties
• Based on the concept of (strong) mathematical induction
• For increasing values of k, check:
  • Base Case: $I(s_0) \land \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \land \neg P(s_k)$
  • Inductive Case: $(\bigwedge_{i=1}^{k+1} T(s_{i-1}, s_i) \land P(s_i)) \land \neg P(s_{k+1})$
• If base case is SAT, return a counter-example
• If inductive case is UNSAT, return TRUE
• Otherwise, continue

Mary Sheeran, Satnam Singh, and Gunnar Stälmárck. Checking safety properties using induction and a SAT-solver. FMCAD 2000
K-Induction Graphically

**Base Case**

- Initial state: $I(s_0)$
- Transition: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$
- Property: $P(s_k)$?

$s_0$ must be an initial state

**Inductive Case**

- Starting state: $s_0$
- Transition: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$
- Property: $P(s_k)$
- Next state: $s_{k+1}$

Arbitrary starting state $s_0$ such that $P(s_0)$ holds
K-Induction: Simple Path

• This approach can be complete over a finite domain
  • requires the simple path constraint
  • each state is distinct from other states in trace
• If simple path is UNSAT, then we can return true
K-Induction: Simple Path

- This approach can be complete over a finite domain
  - requires the simple path constraint
  - each state is distinct from other states in trace
- If simple path is UNSAT, then we can return true

```
\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \]
```

Without simple path, inductive step could get:

```
\[ S_2 \rightarrow S_2 \rightarrow \ldots \rightarrow S_2 \rightarrow S_3 \]
```

Why?

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \]

\[ : \text{not equal} \]
K-Induction Observation

• Crucial observation
  • Does not depend on direct computation of reachable state space

• Beginning of “property directed” techniques
  • We do not need to know the exact reachable states, as long as we can guarantee they meet the property
  • “Property directed” is associated with a family of techniques that build inductive invariants automatically
Inductive Invariants

• The goal of most modern model checking algorithms
• Over finite-domain, just need to show that algorithm makes progress, and it will eventually find an inductive invariant
  • In the worst case, the reachable states are themselves an inductive invariant
  • Hopefully there’s an easier to find inductive invariant that is sufficient

• Inductive Invariant: $II$
  • $Init(s) \Rightarrow II(s)$
  • $T(s, s') \land II(s) \Rightarrow II(s')$
  • $II(s) \Rightarrow P(s)$
Advanced Algorithms

• Interpolant-based model checking
  • Constructs an over-approximation of the reachable states
  • Terminates when it finds an inductive invariant or a counterexample

• IC3 / PDR
  • Computes over (under) approximations of forward (backward) reachable states
  • Refines approximations by guessing relative inductive invariants
  • Terminates when it finds an inductive invariant or a counterexample
Building Blocks: Approximations

• Problems
  • Explicit reachability computation (e.g. BDDs) is difficult
  • Inductive invariants are difficult to find

• Solution (motivation for approximations)
  • Build approximations of reachable states
  • Iteratively refine it until inductive
What is an approximation?

• Actual reachable state set: $R$

• Over-approximation, $O: R \rightarrow O$
  • Proofs on over-approximation holds
  • Counterexamples can be spurious

• Under-approximation, $U: U \rightarrow R$
  • Proofs on under-approximation can be spurious
  • Counterexamples are real
Craig Interpolation

• Given an unsatisfiable formula, $A \land B$

• Craig Interpolant, $I$
  • $A \rightarrow I$
  • $I \land B$ is UNSAT
  • $V(I) \subseteq V(A) \cap V(B)$
    • Where $V$ returns the free variables (uninterpreted constants) of a formula

• We can use interpolants as over-approximations of $A$
Obtaining Craig Interpolants

• Mechanical over SAT
  • Label clauses in the proof
  • Some straightforward post-processing

• Non-trivial for SMT
  • But there are solvers that support it
    • MathSAT
    • Smt-Interpol
    • CVC4 – through SyGuS

K. L. McMillan, Interpolation and SAT-based Model Checking, CAV 2003
Interpolant-based Model Checking

• Big picture
  • Perform BMC
  • Iteratively compute and refine an over-approximation of states reachable in $k$ steps
  • If it becomes inductive, you’re done
Interpolants for Abstraction from BMC Run

• Obtain interpolant, $I$, from an unsat BMC run with $A$ and $B$ as shown below

• Useful properties
  • $I$ over-approximates $A$, i.e. states reachable in one-step from Init: $A \rightarrow I$
  • There are no states reachable in $k - 1$ steps from $I$ that violate the property: $I \land B$ UNSAT
  • $I$ only contains symbols from one time step (time 1): $V(I) \subseteq V(A) \cap V(B)$

$\textbf{Init} \land T(s_0, s_1)$

$T(s_1, s_2) \land \cdots \land T(s_{k-1}, s_k) \land \neg P(s_k)$

From UNSAT $A \land B$, Craig Interpolant, $I$: $A \rightarrow I$

$I \land B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$
Interpolant-based Model Checking

```
if check(Init ∧ T(s₀, s₁) ∧ (¬P(s₀) ∨ ¬P(s₁)))
    return False

k = 2

R = Init

while True
    A := R ∧ T(s₀, s₁), B := ¬P(s_k) ∧ ∧_{i=1}^{k-1} T(s_i, s_{i+1})
    if check(A ∧ B)
        if R == Init
            return False
        else
            R = Init
            k++
    else
        I = get_interpolant()
        R = R ∨ I[1/0] // map symbols at 1 to symbols at 0
        if ¬check(R ∧ T(s₀, s₁) ∧ ¬R)
            return True
```

- **Base case**: Check if \( s₀ \) or \( s₁ \) violate \( P \).
- **Initialize \( R \) to the initial states.**
- **\( A \) = set of states reachable in 1 step from \( R \).**
- **\( B \) = Represents a violation of the property \( P \) in \( K-1 \) steps from the states represented by \( A \).**
- **Check to see if \( P \) is violated is \( K \) steps from \( R \).**
- **If \( A \) and \( B \) is UNSAT, we find an interpolant \( I \).** Recall that \( I \) over-approximates \( A \), i.e., states reachable in one-step from \( R \): \( A \rightarrow I \). Also, there are no states reachable in \( k - 1 \) steps from \( I \) that violate the property: \( I \land B \) UNSAT.
- **Otherwise, increment, reset \( R \) to Init and restart. We may have found a spurious counterexample.**
- **We reached a fixed point where \( R \) is not changing. We found an invariant and proved the property.**
- **Check to see if \( R \land T(s₀, s₁) \rightarrow R \) is valid. I.e., check to see if \( R \land T(s₀, s₁) \land \neg R \) is SAT. If UNSAT, the validity check holds which means the transition function will not grow \( R \).**
Interpolant-based Model Checking Example

• Check to see if initial states or states reachable in 1 step violate P

Init: S0
Bad: P = ¬S9

if check(Init ∧ T(s₀, s₁) ∧ (¬P(s₀) ∨ ¬P(s₁)))
return False
Interpolant-based Model Checking Example

- Check to see if initial states or states reachable in 1 step violate $P$

Init: $S_0$
Bad: $P = \neg S_9$

```python
if check(Init \land T(s_0, s_1) \land (\neg P(s_0) \lor \neg P(s_1)))
    return False
```
Interpolant-based Model Checking Example

• $k = 2$

$k = 2; R = \text{Init}$
while True
  $A := R \land T(s_0, s_1), B := \neg P(s_k) \land \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$
  if check($A \land B$)
Interpolant-based Model Checking Example

• Start – can’t violate in 2 steps

\[ R = \text{Init} \]
while True
\[
A := R \land T(s_0, s_1), B := \neg P(s_k) \land \land_{i=1}^{k-1} T(s_i, s_{i+1})
\]
if check(\(A \land B\))
Interpolant-based Model Checking Example

• $k = 2$

\[
I = \text{get_interpolant}() \\
R = R \lor I[1/0] \quad // \text{map symbols at 1 to symbols at 0} \\
\text{if} \ \neg \text{check}(R \land T(s_0, s_1) \land \neg R) \\
\text{return} \ True
\]

From UNSAT $A \land B$, Craig Interpolant, $I$:

$A \rightarrow I$

$I \land B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$
Interpolant-based Model Checking Example

- $k = 2$

```
while True
    A := R \land T(s_0, s_1),
    B := \neg P(s_k) \land \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})
    if check(A \land B)
```

From UNSAT $A \land B$, Craig Interpolant, $I$:
- $A \rightarrow I$
- $I \land B$ is UNSAT
- $V(I) \subseteq V(A) \cap V(B)$
Interpolant-based Model Checking Example

- \( k = 2 \)

\[
I = \text{get\_interpolant}() \\
R = R \lor I[1/0] \quad // \text{map symbols at 1 to symbols at 0} \\
\text{if } \neg \text{check}(R \land T(s_0,s_1) \land \neg R) \\
\text{return True}
\]

From UNSAT \( A \land B \), Craig Interpolant, \( I \):
\[
A \rightarrow I \\
I \land B \text{ is UNSAT} \\
V(I) \subseteq V(A) \cap V(B)
\]
Interpolant-based Model Checking Example

- \( k = 2 \)

\[ I = \text{get_interpolant}() \]
\[ R = R \lor I[1/0] \quad // \text{map symbols at 1 to symbols at 0} \]
\[ \text{if} \ \neg \text{check}(R \land T(s_0,s_1) \land \neg R) \]
\[ \text{return True} \]

From UNSAT \( A \land B \), Craig Interpolant, \( I \): 
\[ A \rightarrow I \]
\[ I \land B \text{ is UNSAT} \]
\[ V(I) \subseteq V(A) \cap V(B) \]

R: over-approx
Bad: \( P = \neg S_9 \)
Interpolant-based Model Checking Example

- $k = 2$, can reach $S_9$ in 2 steps from $R$

```
if check($A \land B$)
    if $R == \text{Init}$
        return False
    else
        $R = \text{Init}$
        $k++$
```
Interpolant-based Model Checking Example

• $k = 3$, restart with $R = \text{Init}$ and increment $K$

if check($A \land B$)
  if $R == \text{Init}$
    return False
  else
    $R = \text{Init}$
    $k++$

R: over-approx
Bad: $P = \neg S_9$
Interpolant-based Model Checking Example

• $k = 3$, restart with $R = \text{Init}$ and increment $K$

$$R: \text{over-approx}$$

$$\text{Bad: } P = \neg S_9$$

$R = \text{Init}$

while True

$$A := R \land T(s_0, s_1), B := \neg P(s_k) \land \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$$

if check($A \land B$)
Interpolant-based Model Checking Example

- $k = 3$

\[ I = \text{get\_interpolant()} \]
\[ R = R \lor I[1/0] \quad // \text{map symbols at 1 to symbols at 0} \]
\[ \text{if } \neg \text{check}(R \land T(s_0,s_1) \land \neg R) \]
\[ \text{return True} \]

R: over-approx
Bad: $P = \neg S_9$

From UNSAT $A \land B$, Craig Interpolant, $I$:
- $A \rightarrow I$
- $I \land B$ is UNSAT
- $V(I) \subseteq V(A) \cap V(B)$
Interpolant-based Model Checking Example

- $k = 3$

```
R: over-approx
Bad: P = ¬S9

R = Init
while True
    A := R ∧ T(s_0, s_1), B := ¬P(s_k) ∧ ∨_{i=1}^{k-1} T(s_i, s_{i+1})
    if check(A ∧ B)
```
Interpolant-based Model Checking Example

- $k = 3$, interpolant guarantees property not violated in $k-1 \rightarrow 2$ steps

From UNSAT $A \land B$, Craig Interpolant, $I$:

$A \rightarrow I$

$I \land B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$
Interpolant-based Model Checking Example

• Terminate with True! We reached a fixed point!

R: over-approx
Bad: P = \neg S9

\text{if } \neg \text{check}(R \land T(s_0, s_1) \land \neg R) \text{ return True}
Interpolant-based model checking

• Advantages
  • Approximate reachability
  • Clever refinements

• Disadvantages
  • Requires unrolling (can become expensive)
  • Needs to restart every time $k$ is incremented
  • Refinements are clever, but not directly targeting induction