Part I: Basics
- Kripke structures as models of computation
- CTL, LTL and property patterns
- CTL model checking and counterexample generation
- Techniques
  - Symbolic (BDD and SAT)
  - Explicit (reachability and non-termination)
- State of the Art Model Checkers

Overview of Automated Verification
- SW/HW Artifact
- Correctness properties
- Translation
- Model Checker
- Temporal logic
- Yes/No + Counter-example

Computation Tree Logic (CTL)
CTL: Branching time propositional temporal logic
Model: a tree of computation paths
Example:

Kripke Structure

Tree of computation

Models: Kripke Structures
Conventional state machines
- $K = <V, S, s_0, I, R>$
- $V$ is a (finite) set of atomic propositions
- $S$ is a (finite) set of states
- $s_0 \in S$ is a start state
- $I: S \rightarrow 2^V$ is a labeling function that maps each state to the set of propositional variables that hold in it
- Alternatively: a set of interpretations specifying which propositions are true in each state
- $R \subseteq S \times S$ is a transition relation.
Propositional Variables

- Fixed set of atomic propositions \( \{p, q, r\} \)
- Atomic descriptions of a system
  - "Printer is busy"
  - "There are currently no requested jobs for the printer"
  - "Conveyor belt is stopped"
- Should not involve time!

CTL: Computation Tree Logic

- Propositional temporal logic
  - Allows explicit quantification over possible futures

Syntax:

- \( \text{True} \) and \( \text{False} \) are CTL formulae;
- Atomic variables are CTL formulae;
- If \( \varphi \) and \( \psi \) are CTL formulae,
  - then so are: \( \neg \varphi \), \( \varphi \land \psi \), \( \varphi \lor \psi \)
- \( \text{EX} \varphi \): \( \varphi \) holds in some next state
- \( \text{EF} \varphi \): along some path, \( \varphi \) holds in a future state
- \( \text{E} [\varphi \lor \psi] \): along some path, \( \varphi \) holds until \( \psi \) holds
- \( \text{EG} \varphi \): along some path, \( \varphi \) holds in every state

- Universal quantification: \( \text{AX} \varphi \), \( \text{AF} \varphi \), \( \text{A} [\varphi \lor \psi] \), \( \text{AG} \varphi \)

Examples

- \( \text{EX} \varphi \) (exists next)
- \( \text{AX} \varphi \) (all next)
- \( \text{EG} \varphi \) (exists global)
- \( \text{AG} \varphi \) (all global)

Examples (Cont’d)

- \( \text{EF} \varphi \) (exists future)
- \( \text{AF} \varphi \) (all future)
- \( \text{E} [\varphi \lor \psi] \) (exists until)
- \( \text{A} [\varphi \lor \psi] \) (all until)

CTL Examples

Properties that hold:
- \( \text{AX} \text{busy}(s_0) \)
- \( \text{EG} \text{busy}(s_2) \)
- \( \text{A} (\text{req} \lor \text{busy})(s_3) \)

Properties that fail:
- \( \text{AX} \text{req} \lor \text{busy})(s_3) \)

Some Statements To Express

- An elevator can remain idle on the third floor with its doors closed
- When a request occurs, it will eventually be acknowledged
- A process is enabled infinitely often on every computation path
- A process will eventually be permanently deadlocked
- Action \( x \) precedes \( p \) after \( q \)

> Note: hard to do correctly. See later on helpful techniques
A set of connectives is adequate if all formulas can be expressed using it.

\( \varnothing \) is often omitted since we always talk about the same Kripke structure

\( E_s \Rightarrow T \) means that formula \( \varphi \) is true in state \( s \). \( K \)

\( \pi = \pi^0 \pi^1 \ldots \) is a path

\( \pi^0 \) is the current state (root)

\( \pi^{i+1} \) is \( \pi^i \)'s successor state. Then,

- \( A \varphi = \forall \pi \cdot \pi^i = \varphi \)
- \( E \varphi = \exists \pi \cdot \pi^i = \varphi \)
- \( A F \varphi = \forall \pi \cdot \exists \pi^i \cdot A \varphi \)
- \( E F \varphi = \exists \pi \cdot \exists \pi^i \cdot E \varphi \)
- \( A \varphi U \psi = \forall \pi \cdot \exists \pi^i \cdot \pi^i = \psi \land \forall \pi^j \cdot 0 \leq j < i \Rightarrow \pi^i = \varphi \)
- \( E \varphi U \psi = \exists \pi \cdot \exists \pi^i \cdot \exists \pi^j \cdot \pi^i = \psi \land \forall \pi^k \cdot 0 \leq j < i \Rightarrow \pi^j = \varphi \)

**Semantics of CTL**

\( K, s = T \) means that formula \( \varphi \) is true in state \( s \). \( K \)

For reasoning about complete traces through the system

- Allows to make statements about a trace

**Relationship Between CTL Operators**

- \( A \varphi U A \varphi \) = \( E \varphi U A \varphi \)
- \( A \varphi = A[true \ U \ \varphi] \) \( \land \) \( A \varphi U \psi \) = \( \psi \lor (\varphi \land A \varphi U \psi) \)

**Adequate Sets**

Def. A set of connectives is adequate if all connectives can be expressed using it.

\( \{\land, \lor, \Rightarrow, \neg\} \) is adequate for propositional logic:

\( \Rightarrow \lor \land = \neg (\neg \land \land) \)

**Theorem.** The set of operators \( \{\land, \lor, \Rightarrow, \neg\} \) together with \( EX, EG, \) and \( EU \) is adequate for CTL

\( \Rightarrow \) e.g., \( A F (a \lor AX b) = \neg EG \land (a \lor AX b) = \neg EG (\neg a \land EX \neg b) \)

**EU describes reachability**

**EG = non-termination (presence of infinite behaviours)**

**Sublanguages of CTL**

- A CTL formula is in ACTL if it uses only universal temporal connectives \( (AX, AF, AU, AG) \) with negation applied to the level of atomic propositions

- Also called "universal" CTL formulas

- e.g., \( E [p \ U AX \neg q] \)

- ECTL: uses only existential temporal connectives \( (EX, EF, EU, EG) \) with negation applied to the level of atomic propositions

- Also called "existential" CTL formulas

- e.g., \( E [p \ U EX \neg q] \)

- CTL formulas not in ECTL \( \lor \) ACTL are called “mixed”

- e.g., \( E [p \ U AX \neg q] \) and \( A [p \ U EX \neg q] \)

**LTL Syntax**

- If \( \varphi \) is an atomic propositional formula, it is a formula in LTL

- If \( \varphi \) and \( \psi \) are LTL formulas, so are \( \varphi \land \psi, \varphi \lor \psi, \neg \varphi, \varphi \lor (\psi U (\varphi \land \varphi)), X \varphi \) (next), \( F \varphi \) (eventually), \( G \varphi \) (always)

- Interpretation: over computations \( \pi: \omega \Rightarrow 2V \) which assigns truth values to the elements of \( V \) at each time instant

- \( \pi^i = X \varphi \iff \pi^{i+1} = \varphi \)

- \( \pi^i = G \varphi \iff \forall \pi^i \cdot \pi^i = \varphi \)

- \( \pi^i = F \varphi \iff \exists \pi^{i+1} \cdot \pi^{i+1} = \varphi \)

- \( \pi^i \models \varphi U \psi \iff \exists \pi^j \cdot 0 \leq j < i \Rightarrow \pi^j = \varphi \land \pi^i = \psi \)

Here, \( \pi^i \) is \( i \)th state on a path
Properties of LTL

¬ X φ = X ¬ φ
F φ = true U φ
G φ = ¬ F ¬ φ
G φ = φ ∧ X G φ
F φ = φ ∨ X F φ
φ W ψ = G φ ∨ (φ U ψ) (weak until)

A property holds in a model if it holds on every path emanating from the initial state.

Comparison between LTL and CTL

Syntactically: LTL is simpler than CTL
Semantically: incomparable!
- CTL formula EF φ (reachability) not expressible in LTL
- LTL formula F G φ not expressible in CTL
  - What about AF AG φ?
  - Has different interpretation on the following state machine:

LTL and CTL coincide if the model has only one path!

Expressing Properties in LTL

- Good for safety (G ¬) and liveness (F) properties
- Express:
  - When a request occurs, it will eventually be acknowledged
  - Each path contains infinitely many q's
  - At most a finite number of states in each path satisfy ¬q (or property q eventually stabilizes)
  - Action s precedes p after q

  > Note: hard to do correctly. See later on helpful techniques

Property Patterns: Motivation

- Temporal properties are not always easy to write or read
  - e.g., G (q ∧ ¬r ∧ F r) ⇒ (p ⇒ (¬r U (s ∧ ¬r)) U r)
  - Meaning:
    - p triggers s between q (e.g., end of system initialization) and r (start of system shutdown)
- Many useful properties are specifiable in both CTL and LTL
  - e.g., Action q must respond to action p:
    - CTL: AG (p ⇒ AF q)
    - LTL: G (p ⇒ F q)
  - e.g., Action s precedes p after q
    - CTL: A[¬q U (q ∧ A[¬p U s])]
    - LTL: [¬q U (q ∧ [¬p U s])]

Pattern Hierarchy

http://patterns.projects.cis.ksu.edu/

Developers: Dwyer, Avrunin, Corbett

Goal: specifying and reusing property specifications for model-checking

Absence: A condition does not occur within a scope
Existence: A condition must occur within a scope
Universality: A condition occurs throughout a scope
Response: A condition must always be followed by another within a scope
Precedence: A condition must always be preceded by another within a scope

Pattern Hierarchy: Scopes

Scopes of interest over which the condition is evaluated
Using the System: Example

**Property**
- There should be a dequeue() between an enqueue() and an empty().

**Propositions:** deq, enq, em

**Pattern:** “existence” (of deq)
- Scope: “between” (events: enq, em)
- Look up (S exists between Q and R)
  - CTL: AG (Q A R ⇒ A¬R W (S A¬R))
  - LTL: G (Q A¬R ⇒ ¬R W (S A¬R))

**Result**
- CTL: AG (enq A¬em ⇒ A¬em W (deq A¬em)))
- LTL: G (enq A¬em ⇒ ¬em W (deq A¬em)))

CTL Model-Checking

**Inputs:**
- Kripke structure K
- CTL formula \( \varphi \)

**Assumptions:**
- Finite number of processes
  - Each having a finite number of finite-valued variables
- Finite length of a CTL formula

**Algorithm:**
- Label states of K with subformulas of \( \varphi \) that are satisfied there and working outwards towards \( \varphi \)
- Output states labeled with \( \varphi \)
- Example: EX EG \( (p = E[p U q]) \)

**CTL Model-Checking (Cont’d)**

**EX \( \varphi \):**
- Label any state with EX \( \varphi \) if any of its successors are labeled with \( \varphi \)

**EG \( \varphi \):**
- Label every node labeled with \( \varphi \) by EG \( \varphi \)
- Repeat: remove label EG \( \varphi \) from any state that does not have successors labeled by EG \( \varphi \) until there is no change

Counterexamples

**Explain:**
- Why the property fails to hold
- To disprove that \( \varphi \) holds on all elements of \( S \), produce a single element \( s \in S \) s.t. \( \neg \varphi \) holds on \( s \)
- Counterexamples restricted to universally-quantified formulas
- Counterexamples are paths (trees) from initial state illustrating the failure of property

Generating Counterexamples

Negate the prop. and express using EX, EU, EG
- e.g., AG \( (\varphi = AF \psi) \) becomes EF(\( \varphi \land EG \neg \psi \))
- EX \( \varphi \):
  - find a successor state labeled with \( \varphi \)
- EG \( \varphi \):
  - follow successors labeled with \( \varphi \) until a loop is found

CTL Model-Checking (Cont’d)
Generating Counterexamples (Cont’d)

E[φ U ψ]: remove all states not labeled with φ or ψ, then look for path to ψ

This procedure works only for universal properties

- AX φ
- AG (φ ⇒ AF ψ)
- etc.

Symbolic Model Checking (with BDDs)

- Why?
  - Saves us from constructing a model state space explicitly. Effective "cure" for state space explosion.

- How?
  - Sets of states and the transition relation are represented by formulas. Set operations are defined in terms of formula manipulations

- Data Structures
  - ROBDDs – allow for efficient storage and manipulation of logic formulas

- Example:
  \[ x \land y \]

State Explosion

- How fast do Kripke structures grow?
  - Composing linear number of structures yields exponential growth!

- How to deal with this problem?
  - Symbolic model checking with efficient data structures (BDDs, SAT).
    - Do not need to represent and manipulate the entire model
  - Abstraction
    - Abstract away variables in the model which are not relevant to the formula being checked (Part II of tutorial)
  - 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

Representing Models Symbolically

A system state represents an interpretation (truth assignment) for a set of propositional variables \( V \).
Formulas represent sets of states that satisfy it

- False = \( \bot \), True = \( S \)
- \( req = \{ s_0, s_1 \} \)
- \( busy = \{ s_1, s_3 \} \)
- \( req \lor busy = \{ s_0, s_1, s_3 \} \)

State transitions are described by relations over two sets of variables: \( V \) (source state) and \( V' \) (destination state)

Relation \( R \) is described by disjunction of formulas for individual transitions

Representing Boolean Functions

<table>
<thead>
<tr>
<th>Representation of boolean functions</th>
<th>compact?</th>
<th>satisfy</th>
<th>validity</th>
<th>( \lor )</th>
<th>( \land )</th>
<th>\neg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prop. formulas</td>
<td>often</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Formulas in DNF</td>
<td>sometimes</td>
<td>easy</td>
<td>hard</td>
<td>easy</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Formulas in CNF</td>
<td>sometimes</td>
<td>hard</td>
<td>easy</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Ordered truth tables</td>
<td>never</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
<td>hard</td>
</tr>
<tr>
<td>Reduced OBDDs</td>
<td>often</td>
<td>easy</td>
<td>easy</td>
<td>medium</td>
<td>medium</td>
<td>easy</td>
</tr>
</tbody>
</table>

Model-Checking Techniques (Symbolic)

- **BDD**
  - Express transition relation by a formula, represented as BDD. Manipulate these to compute logical operations and fixpoints
  - Based on very fast decision diagram packages (e.g., CUDD)

- **SAT**
  - Expand transition relation a fixed number of steps (e.g., loop unrolling), resulting in a formula
  - For this unrolling, check whether the property holds
  - Continue increasing the unrolling until error is found, resources are exhausted, or diameter of the problem is reached
  - Based on very fast SAT solvers (e.g., ZChaff)
Model-Checking Techniques (Explicit State)
- Model checking as partial graph exploration
- In practice:
  - Compute part of the reachable state-space, with clever techniques for state storage (e.g., Bit-state hashing) and path pruning (partial-order reduction)
  - Check reachability \((X, U)\) properties “on-the-fly”, as state-space is being computed
  - Check non-termination \((G)\) properties by finding an accepting cycle in the graph

Pros and Cons of Model-Checking
- Often cannot express full requirements
  - Instead check several smaller properties
- Few systems can be checked directly
  - Must generally abstract
- Works better for certain types of problems
  - Very useful for control-centered concurrent systems
    - Avionics software
    - Hardware
    - Communication protocols
  - Not very good at data-centered systems
    - User interfaces, databases

Pros and Cons (Cont’d)
- Largely automatic and fast
- Better suited for debugging
  - ... rather than assurance
- Testing vs model checking
  - Usually, find more problems by exploring all behaviours of a downscaled system than by testing some behaviours of the full system

Some State of the Art Model-Checkers
- SMV, NuSMV, Cadence SMV
  - CTL and LTL model-checkers
  - Based on symbolic decision diagrams or SAT solvers
  - Mostly for hardware
- Spin
  - LTL model-checker
  - Explicit state exploration
  - Mostly for communication protocols
- STeP and PVS
  - Combining model-checking with theorem-proving

Abstraction: the key to scaling up
- Too large to analyze directly
- Small, but possibly not precise enough for conclusive analysis

Part II
Model Checking and Abstraction
Part II: Abstraction
- Defining an Abstract Domain
  - variable elimination, data abstraction, predicate abstraction
- Abstraction for Universal/Existential Properties
  - over- and under-approximations
- Abstraction for Mixed Properties
  - 3-valued abstraction
- Overlapping Abstract Domains
  - Belnap (4-valued) abstraction

Defining an Abstract Domain

\[ \alpha : S \rightarrow S' \]

\[ \gamma : S' \rightarrow 2^S \]

Variable Elimination: Example

Abstraction Function: Variable Elimination
- Partition variables
  - ... into visible and
  - ... and invisible
- Abstract states
  - valuations of visible variables
  - ignore invisible variables
- Abstraction function
  - maps each state to its projection over visible variables

Abstraction Function: Data Abstraction
- Partition the data domain
  - e.g., \{EVN,ODD\}, \{NEG,ZERO,POS\}
- Abstraction maps concrete values to elements of the partition

Abstraction Function: Predicate Abstraction
- Pick a finite set of predicates
  - e.g., \{x > y, y < z\}
- Abstraction groups concrete states based on their valuation to \textit{all} of the predicates

Abstraction Function: Example

\[ x_1 \ x_2 \ x_3 \ x_4 \]

\[
\begin{align*}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
\end{align*}
\]

\[ \alpha \ x_1 \ x_2 \]

\[ \begin{align*}
0 & 0 \\
\end{align*} \]

Group concrete states with identical visible part into a single abstract state

Abstraction Function: Predicate Abstraction

\[ x \ y \ z \]

\[
\begin{align*}
-2 & 1 & 9 \\
-1 & 2 & 8 \\
-3 & 3 & 7 \\
\end{align*}
\]

\[ \alpha \ p_1 \ p_2 \]

\[
\begin{align*}
p_1 : x > y \\
p_2 : y < z \\
\end{align*}
\]

\[ \begin{align*}
0 \ p_2 \\
\end{align*} \]
Abstract Kripke Structure

- Abstract interpretation of atomic propositions
  \[ I'(a, p) = \text{true} \iff \forall s \in \gamma(a), I(s, p) = \text{true} \]
  \[ I'(a, p) = \text{false} \iff \forall s \in \gamma(a), I(s, p) = \text{false} \]

- Abstract Transition Relation (2 choices)
  - Over-Approximation (Existential)
    - Make a transition from an abstract state if at least one corresponding concrete state has the transition.
  - Under-Approximation (Universal)
    - Make a transition from an abstract state if all the corresponding concrete states have the transition.

Over-Approximation (Existential Abstraction)

- \[ R^\exists[DGG97] : (a, b) \in R' \iff \exists s \in \gamma(a) \text{ s.t. } \exists t \in \gamma(b) \text{ and } (s, t) \in R \]
- This ensures that \( K' \) is an over-approximation of \( K \), or \( K' \) can match all behaviors of \( K \).

Computing Over-Approximation

- \[ R^\exists[DGG97] : (a, b) \in R' \iff \exists s \in \gamma(a) \text{ s.t. } \exists t \in \gamma(b) \text{ and } (s, t) \in R \]
- This ensures that \( K' \) is an over-approximation of \( K \), or \( K' \) can match all behaviors of \( K \).

Preservation via Over-Approximation

Let \( \phi \) be a universal temporal formula (ACTL, LTL)
Let \( K' \) be an over-approximating abstraction of \( K \)

Preservation Theorem

\[ K' \vDash \phi \implies K \vDash \phi \]

Converse does not hold

\[ K' \nvdash \phi \] does not imply \( K \nvdash \phi \) !!!
\( K' \) may have extra behaviors

Under-Approximation (Universal Abstraction)

- \[ R^\forall[DGG'97] : (a, b) \in R' \iff \forall s \in \gamma(a), \exists t \in \gamma(b) \text{ and } (s, t) \in R \]
- This ensures that \( K' \) is an under-approximation of \( K \), or \( K \) can match all behaviors of \( K' \).

Computing Under-Approximation
Preservation via Under-Approximation

Let $\varphi$ be an existential temporal formula (ECTL)
Let $K'$ be an under approximating abstraction of $K$

Preservation Theorem

$K' \models \varphi$ implies $K \models \varphi$

Converse does not hold

$K' \not\models \varphi$ does not imply $K \not\models \varphi$ !!!

$K'$ may miss some behaviors

Part II: Abstraction

Defining an Abstract Domain

- variable elimination, data abstraction, predicate abstraction

Abstraction for Universal/Existential Properties

- over- and under-approximations

Abstraction for Mixed Properties

- 3-valued abstraction

Overlapping Abstract Domains

- Belnap (4-valued) abstraction

3-Valued Kripke Structures

- Kripke structures extended to 3 valued logic

- Propositions can be
  - True, False, or Unknown

- Transitions
  - possible: \( \bot \)
  - necessary and possible: \( t \)
  - impossible: \( f \)

Which abstraction to use?

<table>
<thead>
<tr>
<th>Property Type</th>
<th>Expected Result</th>
<th>Abstraction to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal (ACTL, LTL)</td>
<td>True</td>
<td>Over-</td>
</tr>
<tr>
<td>Existential (ECTL)</td>
<td>False</td>
<td>Under-</td>
</tr>
</tbody>
</table>

But what about mixed properties?!

3-Valued Kleene Logic

Information Ordering

Truth Ordering

\[
\begin{align*}
t \land \bot &= \bot \\
 t \lor \bot &= t \\
 -t &= f \\
 -\bot &= \bot
\end{align*}
\]

3-Valued Abstraction

3-Valued Abstraction
**Example Revisited (3-Val Abstraction)**

- Usual semantics of temporal operators
- BUT connectives $\land, \lor, \neg$ are interpreted in 3-Valued Logic

\[(EX \, \neg p)(s_0) = t \]
\[(EX \, q)(s_0) = \bot \]
\[(EX \, p \land q)(s_0) = f \]

**Model-Checking with 3 Values**

- $p \lor q$ is true
- $(EX \neg p)(s_0) = \bot$
- $(EX \neg p \land q)(s_0) = f$

**Preservation via 3-Valued Abstraction**

Let $\varphi$ be a temporal formula (CTL)
Let $K$ be a 3-valued abstraction of $K$

<table>
<thead>
<tr>
<th>Concrete Information</th>
<th>Abstract MC Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True (t)</td>
<td>$K \models \varphi$</td>
</tr>
<tr>
<td>False (f)</td>
<td>$K \models \neg \varphi$</td>
</tr>
<tr>
<td>Maybe ($\bot$)</td>
<td>$K \models \varphi$ or $K \models \neg \varphi$</td>
</tr>
</tbody>
</table>

Preserves truth and falsity of arbitrary properties!

**Part II: Abstraction**

- Defining an Abstract Domain
  - variable elimination, data abstraction, predicate abstraction
- Abstraction for Universal/Existential Properties
  - over- and under-approximations
- Abstraction for Mixed Properties
  - 3-valued abstraction
- Overlapping Abstract Domains
  - Belnap (4-valued) abstraction

**Example: Coarse Abstract Domain**

- Over-Approximation: $AX (p \lor \neg p)$ is inconclusive
- Under-Approximation: $EX (q)$ is true

Goal: make $AX$ conclusive as well, via domain refinement

**Example: Refined Abstract Domain**

- Over-Approximation: $AX (p \lor \neg p)$ is true
- Under-Approximation: $EX (q)$ is inconclusive

Partitioned domain does not work!
Need an overlapping abstract domain!!!
**Example: Overlapping Abstract Domain**

Over-Approximation

- $a_1 ightarrow p, q$
- $a_2 ightarrow q$
- $a_3 ightarrow p$

Under-Approximation

- $a_1 ightarrow p$
- $a_2 ightarrow q$
- $a_3 ightarrow p, q$

**AX** $(p \lor \neg p)$ is true

**EX** $(q)$ is true

---

**Supporting Overlapping Abstract Domains**

- **Goal**
  - As before, want to combine over- and under-approximations to support analysis of mixed properties

- **Problem**
  - 3-valued logic is no longer sufficient
  - Need to deal with 4 types of transitions
    - Over, under, both over- and under-, and neither
  - I.e., under-approx is no longer a subset of over-approx

- **Solution**
  - Use 4-valued Belnap logic

---

**Belnap Logic**

<table>
<thead>
<tr>
<th>Information Ordering</th>
<th>Truth Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>$t \land \bot = \bot$</td>
</tr>
<tr>
<td>$t$</td>
<td>$t \lor \bot = t$</td>
</tr>
<tr>
<td>unknown</td>
<td>$\neg t = f$</td>
</tr>
<tr>
<td>inconsistent</td>
<td>$\neg \bot = \bot$</td>
</tr>
</tbody>
</table>

---

**Belnap Kripke Structures**

- **Kripke structures extended to Belnap logic**
- **Propositions**
  - True, False, or Unknown
- **Transitions**
  - Only under-approximation: $\top$
  - Only over-approximation: $\bot$
  - Both over- and under-: $t$
  - Neither: $f$

---

**MV Logic vs Classical Model Checking**

<table>
<thead>
<tr>
<th>Multi-Valued Model Checking</th>
<th>Classical Model Checking</th>
</tr>
</thead>
</table>
| SW/HW Artifact Correctness | properties
| Finite Model Properties |
| Temporal logic |
| Correct? |
| MV Logic |
| Model Checker |
| Yes/No + Counter-example |

---

**Belnap Kripke Structures**

- $p$
- $q$
- $\bot$
- $\top$

---

**MCMC**

- SW/HW Artifact
- Correctness properties
- Temporal logic
- MV Logic
- Model Checker
- Yes/No + Counter-example
Preservation via Belnap Abstraction

Let $\phi$ be a temporal formula (CTL)
Let $K'$ be a Belnap abstraction of $K$

Preservation Theorem

<table>
<thead>
<tr>
<th>Abstract MC Result</th>
<th>Concrete Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>$K = \phi$</td>
</tr>
<tr>
<td>False</td>
<td>$K = \neg \phi$</td>
</tr>
<tr>
<td>$K = \phi$ or $K = \neg \phi$</td>
<td>$K' = \phi$ and $K' = \neg \phi$</td>
</tr>
</tbody>
</table>

Preserves truth and falsity of arbitrary properties!

Summary

Abstraction is the key to scaling up
1. Choose an abstract domain
   - Variable elimination, data abstraction, predicate abstraction, ...
2. Choose a type of abstraction
   - Over-, Under-, 3Val, Belnap
3. Build an abstract model ($\ldots$)
4. Model-check the property on the abstract model
5. If the result is conclusive, STOP
6. Otherwise, pick a new abstract domain, REPEAT

Next: Software Model Checking and Abstraction

Part III

Software Model Checking

In Our Programming Language...
- All variables are global
- Functions are in-lined
- int is integer
  - i.e., no overflow
- Special statements:
  - skip: do nothing
  - assume(e): if e then skip else abort
  - $x,y= e_1, e_2$: $x,y$ are assigned $e_1, e_2$ in parallel
  - $x=\text{nondet}()$: $x$ gets an arbitrary value
  - goto L1,L2: non-deterministically go to L1 or L2

Software Model Checking

From Programs to Kripke Structures

Program

1: int x = 2;
2: int y = 2;
3: while (y <= 2)
4:   y = y – 1;
5: if (x == 2)
6:   ERROR;

State

<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Property: EF (pc = 5)
Programs as Control Flow Graphs

Program

Labeled CFG

1. int x = 2;
2. int y = 2;
3. while (y <= 2)
4.   y = y – 1;
5. if (x == 2)
6.   ERROR:;

Software Model Checking and Abstraction

Kripke Structure K

Boolean Program BP

Program P

Soundness of Abstraction:
BP abstracts P implies that K' approximates K

Abstract Semantics

Abstract Kripke K'

Outline

- Programming Language
  - syntax and semantics
- Predicate Abstraction for Programs
  - Boolean Programs as intermediate representation
  - Automatic computation of abstraction
- Three abstract semantics of Boolean Programs
  - over-, under-, and Belnap abstractions
- Discovering the “right” abstraction automatically
  - Counterexample-guided abstraction refinement
  - Finding a place to refine
    - counterexample- and proof-guided approaches
  - Discovering new predicates
- Overview of state of the art software MCs

The Running Example

<table>
<thead>
<tr>
<th>Program</th>
<th>Property</th>
<th>Expected Answer</th>
</tr>
</thead>
</table>
| int x = 2;
1. int y = 2;
2. while (y <= 2)
3. y = y – 1;
4. if (x = 2)
5. ERROR:; | EF (pc = 5) | False

Model Checking Software

- Programs are not finite state
  - integer variables
  - recursion
  - unbounded data structures
  - dynamic memory allocation
  - dynamic thread creation
  - pointers
  - ...
- Build a finite abstraction
  - ... small enough to analyze
  - ... rich enough to give conclusive results

CounterExample Guided Abstraction Refinement (CEGAR)

Program

Abstract

Boolean Program

Over-Ap...
An Example Abstraction

Program
1: int x = 2;
2: int y = 2;
3: while (y <= 2)
4:   y = y - 1;
5: if (x == 2)
6:   ERROR:;
7: Eph:
8: b = ch(b, f);
9: if (*)
10:     ERROR:;
11: end:

Abstraction
(with y <= 2)
1: bool b is (y <= 2);
2: if (b)
3:   b = ch(b, f);
4: if (*)
5:   ERROR:;
6: end:

Boolean (Predicate) Programs (BP)

- Variables correspond to predicates
- Usual control flow statements
  - while, if-then-else, goto
- Expressions
  - usual Boolean expressions
    - ch(a, b)
      - if a then b else false
      - if a then true else false
- Parallel Assignment
  - $p_1 = ch(a_1, b_1), \ p_2 = ch(a_2, b_2), \ldots$
  - $b_1 = ch(b_1, \neg b_1), \ b_2 = ch(b_2, b_2, f), \ b_3 = ch(f, f)$

Boolean Program Abstraction

- Update $p = ch(a, b)$ is an approximation of a concrete statement $S$ iff $(a)S(p)$ and $(b)S(\neg p)$ are valid
  - i.e., $y = y - 1$ is approximated by
    - $(x == 2) = ch(x == 2, x != 2)$, and
    - $(y <= 2) = ch(y <= 2, false)$
- Parallel assignment approximates a concrete statement $S$ iff all of its updates approximate $S$
  - i.e., $y = y - 1$ is approximated by
    - $(x == 2) = ch(x == 2, x != 2)$,
    - $(y <= 2) = ch(y <= 2, false)$
- A Boolean program approximates a concrete program iff all of its statements approximate corresponding concrete statements

Detour: Weakest Preconditions

Def. A Hoare Triple

{ $P$ } C $S$ { $Q$ } is a logical statement that holds
when $P$ is true and $Q$ is true after executing $S$.

For any state $s$ that satisfies $P$, if executing statement $C$ on $s$
then terminates with a state $s'$, then $s'$ satisfies $Q$.

Def. The weakest precondition of $C$ with respect to $Q$ is a formula $P$ such that
1. $\{P\} C \{Q\}$
2. for all other $P'$ such that $\{P'\} C \{Q\}$,
   $P' \Rightarrow P$ (P is weaker than P').

Calculating Weakest Preconditions

Assignment (easy)

$WP(x = e, \ Q) = Q[x/e]$ (the weakest precondition, x gets the value of e, thus $Q[x/e]$ is required to hold before x=e is executed)

Examples:
- $WP(x = 0, \ x = y) = (x = 0) \Rightarrow (y = 0)$
- $WP(x = 0, \ x = y + 1) = (x = 0) \Rightarrow (y = 1)$
- $WP(x = 0, \ x = y - 1) = (x = 0) \Rightarrow (y < 2)$
- $WP(x = 0, \ x = y - 2) = (x = 0) \Rightarrow (y < 2)$
- $WP(x = 0, \ x = y - 1) = (x = 0) \Rightarrow (y < 2)$

Computing An Abstract Update

$\text{absUpdate (Statement S, List<Predicates> P, Predicate q)}$

\begin{verbatim}
if (tpQ("m => WP(S,q)") resT = resT ∪ m;
    if (tpQ("m = WP(S,¬q)") resF = resF ∪ m;
return "q = ch(resT, resF)"
\end{verbatim}
absUpdate \( y = y - 1 \), \( P = \{ y \leq 2 \} \), \( q = \{ y \leq 2 \} \)

\[ y = y - 1; \]

\[ (y \leq 2) = \text{ch} \ (y \leq 2, f) \]

\[ \text{WP}(y = y - 1, \{ y \leq 2 \}) \text{ is } (y - 1) \leq 2 \]

\[ \text{WP}(y = y - 1, \neg \{ y \leq 2 \}) \text{ is } (y - 1) > 2 \]

Theorem Prover Queries:

\[ (y \leq 2) \Rightarrow (y - 1) \leq 2 \quad \checkmark \]

\[ \neg (y \leq 2) \Rightarrow (y - 1) \leq 2 \quad \times \]

\[ (y \leq 2) \Rightarrow (y - 1) > 2 \quad \times \]

\[ \neg (y \leq 2) \Rightarrow (y - 1) > 2 \quad \times \]

Program Abstraction

\[ \begin{align*}
1: & \text{int } x = 2; \\
2: & \text{int } y = 2; \\
3: & \text{while } (y \leq 2) \\
4: & \quad y = y - 1; \\
5: & \quad \text{if } (x == 2) \\
6: & \quad \text{ERROR;}; \\
7: & \end{align*} \]

But what is the semantics of Boolean programs?

BP Semantics: Overview

- **Over-Approximation**
  - Treat “unknown” as non-deterministic
  - Good for establishing correctness of universal properties

- **Under-Approximation**
  - Treat “unknown” as abort
  - Good for establishing failure of universal properties

- **Exact Approximation**
  - Treat “unknown” as a special unknown value
  - Good for verification and refutation
  - Good for universal, existential, and mixed properties

BP Semantics: Over-Approximation

Abstraction

[Diagram showing over-approximation]

Unknown is treated as non-deterministic

BP Semantics: Under-Approximation

Abstraction

[Diagram showing under-approximation]

Unknown is treated as abort

BP Semantics: Exact Approximation

Abstraction

[Diagram showing exact approximation]

Unknown is treated as unknown
Summary: The Three Semantics

Concrete:
\[ y = y - 1; \]

Abstract:
\[ b_1 \text{ is } (y = 2) \]
\[ b_2 \text{ is } (x = 2) \]
\[ b_1 = \text{ch}(b_1, f) \]
\[ b_2 = \text{ch}(b_2, \neg b_2) \]

Over-Approx
\[ b_1 \text{ b}_2 \]

Belnap (Exact)
\[ b_1 \text{ b}_2 \]

Under-Approx
\[ b_1 \neg b_2 \]

Example: Is ERROR Unreachable?

Program:
\begin{align*}
1: & \text{int } x = 2; \\
2: & \text{while } (y <= 2) \\
3: & \quad y = y - 1; \\
4: & \text{if } (x == 2) \\
5: & \quad \text{ERROR}; \\
6: & \end{align*}

Abstract:
\begin{align*}
1: & b = T; \\
2: & \text{while } (b) \\
3: & \quad b = \text{ch}(b, f); \\
4: & \text{if } (*) \\
5: & \quad \text{ERROR}; \\
6: & \end{align*}

CounterExample Guided Abstraction Refinement (CEGAR)

Program:
\begin{align*}
1: \text{int } x = 2; \\
2: \text{int } y = 2; \\
3: \text{while } (y <= 2) \\
4: \quad y = y - 1; \\
5: \text{if } (x == 2) \\
6: \quad \text{ERROR}; \\
7: \end{align*}

Abstract:
\begin{align*}
1: \text{bool } b \text{ is } (y <= 2) \\
2: & b = T; \\
3: \text{while } (b) \\
4: \quad b = \text{ch}(b, \neg b); \\
5: \text{if } (*) \\
6: \quad \text{ERROR}; \\
7: \end{align*}

Example: Is ERROR Unreachable?

Program Abstraction

CEGAR steps
Abstract \rightarrow Translate \rightarrow Check \rightarrow Validate \rightarrow Repeat

CounterExample Guided Abstraction Refinement (CEGAR)

Program:
\begin{align*}
1: \text{int } x = 2; \\
2: \text{int } y = 2; \\
3: \text{while } (y <= 2) \\
4: \quad y = y - 1; \\
5: \text{if } (x == 2) \\
6: \quad \text{ERROR}; \\
7: \end{align*}

Abstract:
\begin{align*}
1: \text{bool } b \text{ is } (y <= 2) \\
2: & b = T; \\
3: \text{while } (b) \\
4: \quad b = \text{ch}(b, \neg b); \\
5: \text{if } (*) \\
6: \quad \text{ERROR}; \\
7: \end{align*}

Example: Is ERROR Unreachable?

Program Abstraction

CEGAR steps
Abstract \rightarrow Translate \rightarrow Check \rightarrow NO ERROR
Using Cex for Refinement

- Using Cex for refinement when proofs are used to guide the refinement.
- Only a part of the proof must be generated.
- No need to validate the counterexample.
- ... unknown steps are already marked in the proof.
- Refinement is not limited to finite linear explanations.

Finding Refinement Predicates

- Recall:
  - Each abstract state is a conjunction of predicates.
  - Each abstract transition corresponds to a program statement.

- Result from a partial proof:
  - Unknown transition $s_1 \rightarrow s_2$.
  - $\{s_1\} C \{s_2\}$

- MC needs to know the validity of $C$.

- New predicate: $\text{WP}(y = y - 1, y > 2)$.

An Example

- $s_1 \rightarrow s_2$ is unknown.
- $p = \{y > 2 \land x = 2\}$
  - $y = y - 1$ \(\land\) $y > 2$ \(\land\) $x = 2$
  - $y > 2$ \(\land\) $x = 2$
- New predicate: $\text{WP}(y = y - 1, y > 2) = y > 3$.

Refinement via Weakest Precondition

- If $s_1 \rightarrow s_2$ corresponds to a conditional statement,
  - Refine by adding the condition as a new predicate.

- If $s_1 \rightarrow s_2$ corresponds to a statement $C$,
  - Find a predicate $p$ in $s_2$ with uncertain value.
  - Refine by adding $\text{WP}(C, p)$.

Summary: Software Model Checking

- SoftMC is an effective technique for analyzing behavioral properties of software systems.
- Based on a combination of static analysis and traditional model checking techniques.
- Abstraction is essential for scalability.
- Boolean programs are used as an intermediate step.
- Different abstract semantics lead to different abs.
- Automatic abstraction refinement enables to find the “right” abstraction incrementally.
Overview of Software Model Checkers

- **Tools:**
  - YASM
  - SLAM
  - BLAST
  - CBMC
  - MAGIC
  - Java PathFinder

Comparison parameters
- Properties
- Types of abstraction
- Model-checking engine
- How refinement is done

Yet Another Software Model-checker

YASM

- [http://www.cs.toronto.edu/~arie/yasm](http://www.cs.toronto.edu/~arie/yasm)
- **Properties:** CTL
- **Abstraction:** Predicate Over- and Under-
- **MC Engine:** Symbolic BDD based
- **Refinement:** CTL Proof based + WP

Main Features of YASM

- Checks real C programs
- Not biased towards verification or refutation
- Sound for both True and False answers
- Can check arbitrary CTL property
  - ... including liveness!
- Handles recursive programs

Current Applications

- **BLAST Benchmarks** [GC06]
  - Device drivers (4K-6K LOC)
  - Parts of OpenSSH (2K-3K LOC)
- **Split OpenSSH (100K LOC)**
  - with UoT Security Group
- **Concurrent “Toy” Programs**
  - Lamport’s Bakery Mutual Exclusion
  - Error detection in NASA RAX [PPV05]
- **Finding livelock bugs**
  - “Can a library routine get stuck?”
  - with B. Cook at Microsoft Research, in progress

SLAM (Microsoft)

- Part of Windows DDK Static Driver Verifier
- **Properties:** Reachability
- **Abstraction:** Predicate over approximation
- **MC Engine:** Symbolic BDD based
- **Refinement:** Symbolic simulation of cexs
- **Key Features:**
  - very robust
  - supports recursion
  - (almost) in production use
Part IV

Usability Issues

**BLAST**
- [http://embedded.eecs.berkeley.edu/blast/](http://embedded.eecs.berkeley.edu/blast/)
- **Properties**: Reachability
- **Abstraction**: Predicate over-approximation
- **MC Engine**: Symbolic BDD based
- **Refinement**: Predicates from a proof of impossibility of a counterexample

**SATABS & CBMC**
- **Properties**: Bounded reachability
- **Abstraction**: Predicate over-approximation
- **MC Engine**: Symbolic SAT based
- **Refinement**: Symbolic simulation of cex + UNSATCORE
- **Key Features**: support for precise machine arithmetic including bit level operations

**MAGIC**
- **Properties**: Automata Simulation
- **Abstraction**: Predicate over-approximation
- **MC Engine**: SAT based
- **Refinement**: Symbolic simulation of cex
- **Key Features**: support for concurrent C modules

**Java PathFinder**
- **Properties**: Reachability
- **Abstraction**: user-provided data abstraction
- **MC Engine**: Explicit state with symbolic execution
- **Refinement**: None
- **Key Features**: support for Java including Objects and Threads

**Usability Issues (Our Work)**

- Obtaining "most interesting" counterexample
- Finding "right" properties
- Correctness properties
- Temporal logic
- Trusting the Yes answer
- Yes/No + counterexample
- Model Checker
- Model Extraction
- SW/HW artifact
- Model of System
- Model Extraction Translation
- Translation
Some of our projects

- Multi-Valued Model Checking
  - Reasoning with partial and inconsistent information

- Software Model Checking
  - Checking behavioral properties of programs

- Understanding Counterexamples
  - Understanding and exploring results of automated analysis

- Temporal Logic Query Checking
  - Computer-aided model exploration

- Vaccum Detection
  - How to trust automated analysis

Dealing with Vaccum: Manual Approach

- Check that antecedent of implication is satisfied in at least one state
  \[ EF (\text{req}) \land AG (\text{req} \implies AF \text{ack}) \]
- Often hard to get right for long properties
- Defeats the purpose of model checking as automatic technique

Our Vaccum Project [GC04]

- Goal: Automated vaccum detection
- Formalize the notion of vaccum
- Create effective algorithms for
  - Identifying the cause of vaccum
  - Producing witnesses to non-vaccum
- Create fast (comparable to model checking in speed and time) implementations
  - For model-checkers based on decision diagrams
    - VQuoT (see demo at FM’06)
  - For SAT-based model-checkers
    - VQuad/Tree (see demo at FM’06)

Towards shortening the cycle

- Check phase
  - Running the model-checker, so want to minimize # of runs
- Analyze phase: time spent by a human
  - Too much evidence – BAD!
    - Hard to build a mental picture
    - Takes too much effort to reach the place of interest
    - May not notice repeated patterns
- Too little evidence – BAD!
  - If there are several reasons for a failure, may want to see all of them
    Ex.: \( f \) and \( g \) fail because BOTH are false
Interactive Explanations

- User can control:
  - Kinds of evidence that get generated
    - i.e., prefer traces that go through the previously explored part of the model
  - Amount of information generated and presented
    - By restricting the scope of exploration: $AG (a \rightarrow AF b)$
  - Time a model-checker spends computing evidence
    - So they can continue exploring it manually

- Advantages:
  - Amount of evidence generated is based on what user is willing to understand
  - Amount of evidence displayed helps identify “interesting” cases and aid with debugging

Navigational choices for witnesses

- Choices
  - explicit (disjunction)
    - which part of property to consider
    - Example: $(EF p) \lor (EG q)$
  - implicit (via EX)
    - which state to pick as a witness?
    - Example: $EX p$

- By default, choice is random, with goal to find shortest witness

Elevator Controller System

Button model

- $r$ - request to move has been generated
- $f$ - request is fulfilled, button can be reset
- $p$ - state of button (pressed or released)

Task 1: Getting the property right

- Attempt 1: $AG \ AF (floor = 3 \land door = open)$
  - No: can get stuck on first floor
- Attempt 2: $AG (floor \neq 1 \rightarrow AF (floor = 3 \land door = open))$
  - No: can get stuck on second floor

- Solution 0: Hope for a sudden revelation!
- Solution 1: Generate all counterexamples
  - Attempt 2:
    - No: can oscillate between first and second floor
  - Attempt 3: $AG (btn3.r \rightarrow AF (floor = 3 \land door = open))$
    - YES!

- Solution 2: specify a strategy to avoid a state where floor=1
  - i.e., can get multiple counterexamples without modifying the property

Task 2: Reducing cognitive overload

- Why: want to stay in the part of the program that is already better understood
  - Designated state Idle
    - $floor=1$, doors are closed, direction is up, state is notMoving

- Strategies:
  - Guide counterexample generator towards such state
  - Keep track of states visited during previous verification
    - And choose those!

- Exploration vs. verification
  - Verification: prefer most familiar part of system
    - Minimize distance to Idle
  - Exploration: prefer least familiar part of system
    - Maximize distance to Idle

Task 3: choosing “best” loop

- Why? (Attempt at “shortest” counterexample)
  - Counterexamples for $EG p$ properties:

- Goal: find “best” loop
  - … around most familiar state (Idle)
  - … most interesting, using loop summaries
How do we do it?

Create: a counterexample as a proof [CG06]
Proofs are great for capturing underlying structure and navigating through it

Navigate:
- By hand
- Automated through strategies

Proof presentation
- In terms of the model (i.e., traces, successors)

Generating proofs is not $$: they are gathered from results of a model-checking run

Proofs-like Witness

Why does $\text{EF} (f \lor r)(s_0)$ hold, i.e., why is $(f \lor r)$ reachable?

Witness

Proof

\[
\exists t \in \mathbb{N} \quad (f \lor r)(t) \land R(s_0, t)
\]

Proof

\[
\text{EF} (f \lor r)(s_0)
\]

\[
\text{EF}^2 (f \lor r)(s_0)
\]

\[
\text{EX EX} (f \lor r)(s_0)
\]

\[
\text{EX} (f \lor r)(s_1)
\]

\[
R(s_1, s_2)
\]

\[
(f \lor r)(s_2)
\]

\[
\text{EF} (f \lor r)(s_0)
\]

Proof

Witness
Visualization Engine

- Produce proof-like counterexamples
- Present proof summaries (“what is going to follow”)
- Visualization strategies
  - Restrict scope of explanation (starting/stopping)
    - Example: EG EF (x ∧ EX x) and want to see witness to EF
  - Starting condition: EF (x ∧ EX x)
  - Stopping condition: x ∧ EX x
- Give state name / variables in state
- Display entire state / only changes
- Verbosity of explanation
  - Proof / English summary
- Forward/backward exploration

KEGVis: Witness view

KEGVis: Proof View

Why Model Understanding

Software (Model) Engineering
- Specification
- Design
- Implementation
- Verification
- Testing

Model Understanding
- Specification
  - Design
- Model

Temporal Logic Query Checking

Computer-Aided Model Understanding

Model Understanding - Structural
- Modules and dependencies
- Design and architectural patterns

Why Model Understanding - Structural
- Modules and dependencies
  - main.c
  - foo.c
  - stuff.c
  - bar.c
**Model Understanding - Behavioural**

- Scenarios
  - sample behaviors

- Properties
  - succinct summaries of behaviors

  "X is an invariant"
  "X is eventually followed by Y"

**Example: Cruise Control System (CCS)**

- Maintains a speed of an automobile
  - Four major modes of operation (indicated by variable \( CC \))
    - \( CC = \text{Off} \) - cruise control is off
    - \( CC = \text{Inactive} \) - cruise control is idle
    - \( CC = \text{Cruise} \) - maintaining the speed of the automobile
    - \( CC = \text{Override} \) - overridden by the user
      - (i.e. brake pedal is pressed)

- Relevant parts of the automobile are modeled as well
  - Ignition, Running, Brake, Throttle, etc

**Query Checking**

- TL Property
  - \( p \) is an invariant
  - \( AG \) \( p \)

- TL Query
  - What is an invariant?
  - \( AG \) ?

  **Propositional Solutions**
  - \( (p \lor q) \land r \)

**Computer-Aided Model Understanding**

- Unguided

- Property Guided

- Sample Templates
  - What are all reachable modes?
    - T: "Mode ___ is reachable"
    - S: How each mode is reached?

  - Where can the system evolve to from mode Off?
    - T: "When Off is reached, mode ___ follows"
    - S: How does this happen?

  - What is known about Ignition, Running, and Brake when CCS is Inactive?
    - T: When mode is Inactive, then ___ (w.r.t. Ignition, Running, Brake)

  - What pairs of modes follow each other
    - T: "When mode ___ is reached, mode ___ follows"
    - S: How does this happen?

**Goals of TLQSolver Project [GCD03]**

1. Extend the language of queries (templates)
2. Enable automated support for scenario generation
3. Build a working implementation
4. Explore software engineering applications
The Language of Queries

- Queries based on arbitrary CTL properties, with multiple occurrence of a placeholder e.g. “what happens twice in a row?”
  \[ EF (\varphi \land EX \varphi) \]

- several different placeholders e.g. “what states can follow each other?”
  \[ EF (\varphi \land EX \varphi) \]

- allow restrictions on placeholders e.g. “what modes can follow each other?”
  \[ EF (\varphi_{\{CC\}} \land EX \varphi_{\{CC\}}) \]

Detour: Multi-Valued Model-Checking

When values form a lattice
‘\textendash’ theory and implementation of all these tools is the same!

Query-Checking as Multi-Valued MC

- Multi-valued model-checker can iterate over such lattices
  \[
  \begin{array}{c}
  \text{true} \\
  \text{p} \\
  \neg p \\
  \text{false}
  \end{array}
  \]
  Lattice of propositional formulas over \{p\} on implication

- Reduce query-checking to multi-valued model-checking!
  \[
  \begin{array}{c}
  \{\text{false}, \text{p}, \neg \text{p}, \text{true}\} \\
  \{\text{p}, \text{true}\} \\
  \{\neg \text{p}, \text{true}\} \\
  \{\text{true}\} \\
  \{\}\n  \end{array}
  \]
  Lattice of possible solutions on set inclusion

TLQSolver

Query-Checking via Multi-Valued Model-Checking

- TLQSolver
  - Query-Checking
    - Convert to MvCTL
    - Use XChek

Generated Witness

Query:

\[ EF((CC=Off) \land EX ?(CC)) \]
Application: Guided Simulation

**Output:**
- one trace: Off ⇒ Inactive ⇒ Cruise ⇒ Override
- sequence of events: @T(Ignition), @T(Running), @T(Button=bCruise), @T(Button=bOff)
- no user input required!

- Guided simulation
  - specify **objective** via a query
  - witness serves as basis for simulation
- Example:
  - goal: \( EF \neg (CC) \)
  - prefer witnesses with largest common prefix

Other Applications

- Invariant discovery
  - e.g., what is true when CCS is in mode Cruise
- Precondition discovery
  - e.g., what guarantees transition from Off to Inactive
- Test case generation
  - Query encodes test coverage criterion
  - A witness is a test-suite achieving this coverage
- Planning
  - Query encodes plan objective
  - A witness is a plan

Current/Future Work

- General query checking too expensive
  - exponential in \# of states of the model
- ... and often “too much”:

  - State based queries can be solved efficiently
  - polynomial in \# of states of the model
- See FM ’06 demo!

Summary

- Model understanding is an integral part of software engineering activities
- Computer-aided understanding is possible with the use of templates
- TLQSolver and Temporal Logic Queries
  - Expressive language of templates
  - Control over what scenarios are generated and displayed
  - Applicable to various software engineering activities
  - Packets of easier query-checking problems

Summary of Part IV

- Vacuity Detection
  - How to trust automated analysis
- Understanding and exploring results of automated analysis

Tutorial Summary

- Part I: Basics
  - Temporal logics (CTL, LTL), model-checking, counter-example generation, symbolic model-checking, state-of-the-art model-checkers
- Part II: Abstraction
  - Over-approximating, under-approximating and Belnap abstractions and properties they preserve
- Part III: Software Model Checking
  - Abstraction-refinement framework, techniques for analyzing programs, building boolean programs, refinement, relationships between abstract and concrete systems, state of the art SoftMCs
- Part IV: Usability Issues
  - Vacuity, understanding counter-examples, model exploration with query-checking
References

[Ch00] W. Chan. “Temporal Logic Queries”. In CAV’00.

Acknowledgements
We thank the model checking group at CMU (Ed Clarke) and the BANDERA project (Matt Dwyer, Corina Pasareanu) for the source of and the inspiration for some of our slides.

References

[GC04] A. Gurfinkel, M. Chechik. “How Vacuous is Vacuous”. In TACAS’04.
[GC06] A. Gurfinkel, M. Chechik. “Why Waste a Perfectly Good Abstraction”. In TACAS’06.
[PPV05] C. Pasareanu, R. Pelanek, W. Visser. “Concrete Search with Abstract Matching and Refinement”. In CAV’05.