Model-Checking: From Hardware to Software

Marsha Chechik Arie Gurfinkel

FM 2006 Tutorial

August 22, 2006

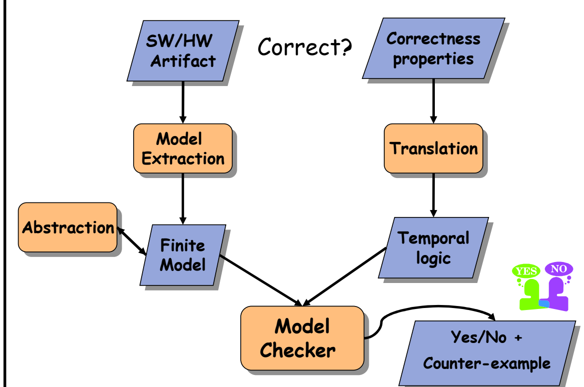
Tutorial Overview

- Part I: Introduction and 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 (time permitting)
  - Vacuity, understanding counter-examples, model exploration with query-checking

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

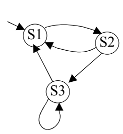


Computation Tree Logic (CTL)

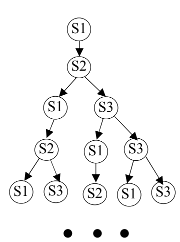
CTL: Branching-time propositional temporal logic

Model - a tree of computation paths

Example:



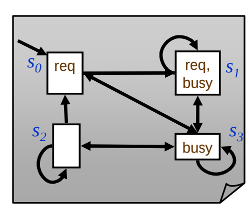
Kripke Structure



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"
  - "Conveyer belt is stopped"
- Should not involve time!

CTL: Computation Tree Logic

propositional temporal logic

allows explicit quantification over possible futures

Syntax:

True **and** ***False*** are CTL formulae;
propositional variables are CTL formulae;

If φ and ψ are CTL formulae,

then so are: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$

EX φ : φ holds in some next state

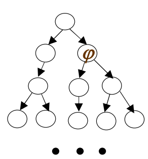
EF φ : along some path, φ holds in a future state

E[φ U ψ] : along some path, φ holds until ψ holds

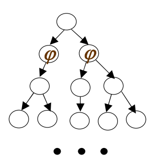
EG φ : along some path, φ holds in every state

- <u>Universal quantification:</u> AX φ, AF φ, A[φ U ψ], AG φ

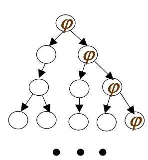
Examples



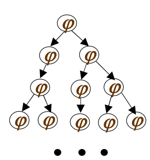
EX φ (exists next)



AX φ (all next)

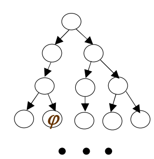


EG φ (exists global)

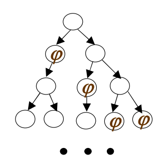


AG φ (all global)

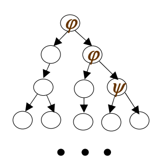
Examples (Cont'd)



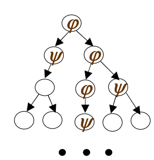
EF φ (exists future)



AF φ (all future)



E[φ U ψ] (exists until)



A[φ U ψ] (all until)

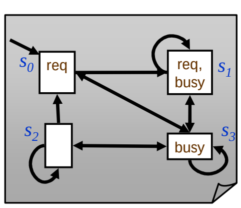
CTL Examples

Properties that hold:

- (AX busy)(s_0)
- (EG busy)(s_3)
- A (req U busy) (s_0)
- E ($\neg$req U busy) (s_1)
- AG (req $\Rightarrow$ AF busy) (s_0)

Properties that fail:

- (AX (req $\vee$ 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 *s* precedes *p* after *q*
  - Note: hard to do correctly. See later on helpful techniques

Semantics of CTL

$K,s \models \varphi$ – means that formula φ is true in state s. K is often omitted since we always talk about the same Kripke structure

- E.g., $s \models p \wedge \neg q$

$\pi = \pi^0 \pi^1 \ldots$ is a path

π^0 is the current state (root)

π^{i+1} is π^i's successor state. Then,

- $AX\ \varphi = \forall \pi \cdot \pi^1 \models \varphi$ $\quad$ $EX\ \varphi = \exists \pi \cdot \pi^1 \models \varphi$
- $AG\ \varphi = \forall \pi \cdot \forall i \cdot \pi^i \models \varphi$ $\quad$ $EG\ \varphi = \exists \pi \cdot \forall i \cdot \pi^i \models \varphi$
- $AF\ \varphi = \forall \pi \cdot \exists i \cdot \pi^i \models \varphi$ $\quad$ $EF\ \varphi = \exists \pi \cdot \exists i \cdot \pi^i \models \varphi$
- $A[\varphi\ U\ \psi] = \forall \pi \cdot \exists i \cdot \pi^i \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \models \varphi$
- $E[\varphi\ U\ \psi] = \exists \pi \cdot \exists i \cdot \pi^i \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \models \varphi$

Relationship Between CTL Operators

$\neg AX\varphi = EX\ \neg\varphi$

$\neg AF\varphi = EG\ \neg\varphi$ $\quad$ $\neg EF\varphi = AG\ \neg\varphi$

$AF\varphi = A[\text{true}\ U\ \varphi]$ $\quad$ $EF\varphi = E[\text{true}\ U\ \varphi]$

$AG\ \varphi = \varphi \wedge AX\ AG\ \varphi$ $\quad$ $EG\ \varphi = \varphi \wedge EX\ EG\ \varphi$

$AF\ \varphi = \varphi \vee AX\ AF\ \varphi$ $\quad$ $EF\ \varphi = \varphi \vee EX\ EF\ \varphi$

$A\ [\text{false}\ U\ \varphi] = E[\text{false}\ U\ \varphi] = \varphi$

$A[\varphi\ U\ \psi] = \neg E[\neg\psi\ U\ (\neg\varphi \wedge \neg\psi)] \wedge \neg EG\ \neg\psi$

$A[\varphi\ U\ \psi] = \psi \vee (\varphi \wedge AX\ A[\varphi\ U\ \psi])$

$E[\varphi\ U\ \psi] = \psi \vee (\varphi \wedge EX\ E[\varphi\ U\ \psi])$

$A[\varphi\ W\ \psi] = \neg E[\neg\psi\ U\ (\neg\varphi \wedge \neg\psi)]$ (weak until)

$E[\varphi\ U\ \psi] = \neg A[\neg\psi\ W\ (\neg\varphi \wedge \neg\psi)]$

Adequate Sets

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

- e.g., $\{\neg, \wedge\}$ is adequate for propositional logic:
  - $a \vee b = \neg(\neg a \wedge \neg b)$

Theorem. The set of operators {false, ¬, ∧} together with EX, EG, and EU is adequate for CTL

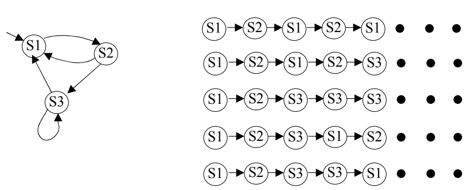
- - e.g., $AF\ (a \vee AX\ b) = \neg\ EG\ \neg\ (a \vee AX\ b) = \neg\ EG\ (\neg a \wedge EX\ \neg b)$
- EU describes reachability
- EG – non-termination (presence of infinite behaviours)

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., $A\ [p\ U\ AX\ \neg q]$
- 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 ∪ ACTL are called "mixed"
  - e.g., $E\ [p\ U\ AX\ \neg q]$ and $A\ [p\ U\ EX\ \neg q]$

Linear Temporal Logic (LTL)

For reasoning about complete traces through the system



Allows to make statements about a trace

LTL Syntax

- If φ is an atomic propositional formula, it is a formula in LTL
- If φ and ψ are LTL formulas, so are $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$, $\varphi\ U\ \psi$ (until), $X\ \varphi$ (next), $F\varphi$ (eventually), $G\ \varphi$ (always)
- Interpretation: over computations $\pi: \omega \Rightarrow 2^V$ which assigns truth values to the elements of V at each time instant
  - $\pi \models X\ \varphi$ iff $\pi^1 \models \varphi$
  - $\pi \models G\ \varphi$ iff $\forall i \cdot \pi^i \models \varphi$
  - $\pi \models F\varphi$ iff $\exists i \cdot \pi^i \models \varphi$
  - $\pi \models \varphi\ U\ \psi$ iff $\exists i \cdot \pi^i \models \psi \wedge \forall j \cdot 0 \leq j < i \Rightarrow \pi^j \models \varphi$
  - Here, π^i – i'th state on a path

Properties of LTL

$\neg X\varphi = X\neg\varphi$

$F\varphi = \text{true } U\,\varphi$

$G\varphi = \neg F\neg\varphi$

$G\varphi = \varphi \wedge X\,G\,\varphi$

$F\varphi = \varphi \vee X\,F\,\varphi$

$\varphi\, W\, \psi = G\,\varphi \vee (\varphi\, U\, \psi)$ (weak until)

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

Expressing Properties in LTL

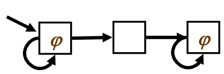
- Good for safety ($G\neg$) and liveness (F) properties
- Express:
  - When a request occurs, it will eventually be acknowledged
  - Each path contains infinitely many *q*'s
  - At most a finite number of states in each path satisfy $\neg q$ (or property *q* eventually stabilizes)
  - Action *s* precedes *p* after *q*
    - Note: hard to do correctly. See later on helpful techniques

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:



  - AF AG φ is false
  - F G φ is true

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

Property Patterns: Motivation

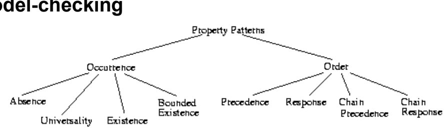
- Temporal properties are not always easy to write or read
  - e.g., $G\,((q \wedge \neg r \wedge F\,r) \Rightarrow (p \Rightarrow (\neg r\ U\ (s \wedge \neg r))\ U\ r)$
  - Meaning:
    - *p* triggers *s* between *q* (e.g., end of system initialization) and *r* (start of system shutdown)
- Many useful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

Pattern Hierarchy

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

Developers: Dwyer, Avrunin, Corbett

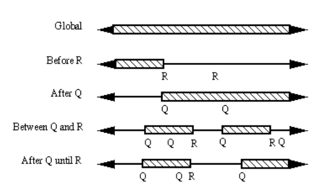
Goal: specifying and reusing property specifications for model-checking



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 ($Q \wedge \neg R \Rightarrow$ A[$\neg R$ W ($S \wedge \neg R$)])
  - ➢ LTL: G ($Q \wedge \neg R \Rightarrow$ ($\neg R$ W ($S \wedge \neg R$)))

➲ Result
  - ➢ CTL: AG (enq $\wedge \neg$ em $\Rightarrow$ A[$\neg$ em W (deq $\wedge \neg$ em)])
  - ➢ LTL: G (enq $\wedge \neg$ em $\Rightarrow$ ($\neg$ em W (deq $\wedge \neg$ em)))

CTL Model-Checking

➲ Inputs:
- ✎ Kripke structure *K*
- ✎ CTL formula φ

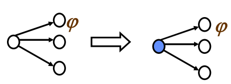
➲ Assumptions:
- ✎ Finite number of processes
  - ➢ Each having a finite number of finite-valued variables
- ✎ Finite length of a CTL formula

➲ Algorithm:
- ✎ Label states of *K* with subformulas of φ that are satisfied there and working outwards towards φ.
- ✎ Output states labeled with φ

Example: EX EG ($p \Rightarrow$ E[p U q])

CTL Model-Checking (Cont'd)

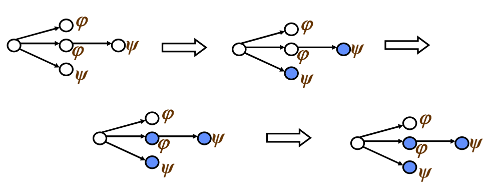
EX φ
- ✎ Label any state with EX φ if any of its successors are labeled with φ



E [φ U ψ]
- ✎ If any state s is labeled with ψ, label it with E[φ U ψ]
- ✎ Repeat:

  label any state with E[φ U ψ] if it is labeled with φ and at least one of its successors is labeled with E[φ U ψ]

  until there is no change

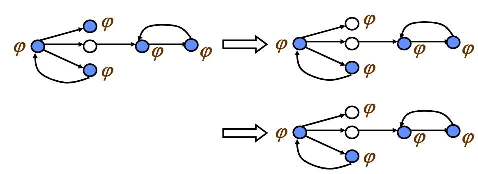


CTL Model-Checking (Cont'd)

EG φ
- ✎ Label every node labeled with φ by EG φ
- ✎ Repeat:

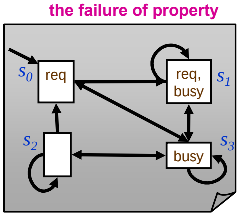
  remove label EG φ from any state that does not have successors labeled by EG φ

  until there is no change

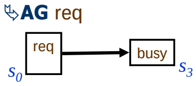


Counterexamples

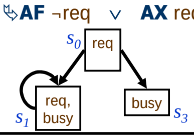
Explain:
- ✎ Why the property fails to hold
- ✎ to disprove that ϕ holds on all elements of *S*, produce a single element $s \in S$ s.t. $\neg\phi$ holds on *s*.
  - ➢ counterexamples restricted to universally-quantified formulas
  - ➢ counterexamples are paths (trees) from initial state illustrating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



✎ AG req

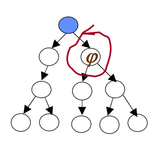


✎ AF $\neg$req $\vee$ AX req

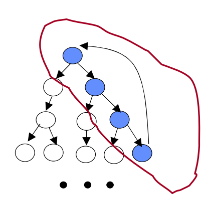


Generating Counterexamples

Negate the prop. and express using EX, EU, EG
- ✎ e.g., AG ($\varphi \Rightarrow$ AF ψ) becomes EF($\varphi \wedge$ EG $\neg$ ψ)

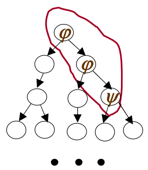


EX φ:
find a successor state labeled with φ



EG φ:
follow successors labeled with EG φ until a loop is found

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

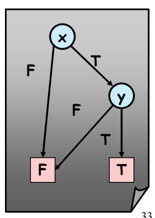
- **AX φ**
- **AG ($\varphi \Rightarrow$ AF ψ)**
- **etc.**

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**

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∧y

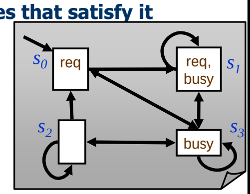


Representing Models Symbolically

A system state represents an interpretation (truth assignment) for a set of propositional variables ***V***

Formulas represent sets of states that satisfy it

- ***False* = ∅, *True* = *S***
- **req – set of states in which req is true – {s_0, s_1}**
- **busy – set of states in which busy is true – {s_1, s_3}**
- **req ∨ 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)

- **Transition (s_2, s_3) is ¬req ∧ ¬ busy ∧ ¬req′ ∧ busy′**
- **Relation *R* is described by disjunction of formulas for individual transitions**

Representing Boolean Functions

Representation of boolean functions	compact?	satisf'ty	validity	∧	∨	¬
Prop. formulas	often	hard	hard	easy	easy	easy
Formulas in DNF	sometimes	easy	hard	hard	easy	hard
Formulas in CNF	sometimes	hard	easy	easy	hard	hard
Ordered truth tables	never	hard	hard	hard	hard	hard
Reduced OBDDs	often	easy	easy	medium	medium	easy

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

37

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

38

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

39

Some State of the Art Model-Checkers

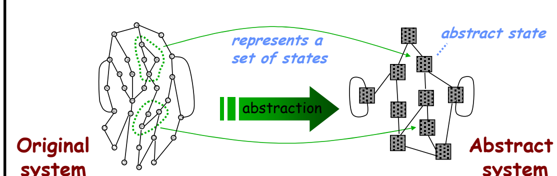
- SMV, NuSMV, Cadence SMV
  - CTL and LTL model-checkers
  - Based on symbolic decision diagrams or SAT solvers
  - Mostly for hardware
- Spin
  - LTL model-checker
  - Explicit state exploration
  - Mostly for communication protocols
- STeP and PVS
  - Combining model-checking with theorem-proving

40

Part II

Model Checking and Abstraction

Abstraction: the key to scaling up



Too large to analyze directly

Small, but possibly not precise enough for conclusive analysis

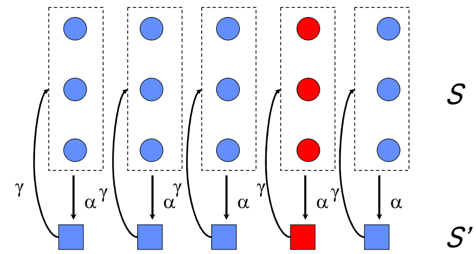
42

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**

43

Defining an Abstract Domain



Abstraction $\alpha : S \rightarrow S'$ Concretization $\gamma : S' \rightarrow 2^S$

44

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**

45

Variable Elimination: Example

$x1$	$x2$	$x3$	$x4$
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1

$\xrightarrow{\alpha}$

$x1$	$x2$
0	0

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

46

Abstraction Function: Data Abstraction

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

$x1$	$x2$	$x3$
1	-3	0
3	-4	0
6	-1	0
…	…	…

$\xrightarrow{\alpha}$

$x1$	$x2$	$x3$
POS	NEG	ZERO

47

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 *all* of the predicates**

***p1*: x > y**
***p2*: y < z**

x	y	z
-2	1	9
-1	2	8
-3	3	7
…	…	…

$\xrightarrow{\alpha}$

$p1$	$p2$
0	1

48

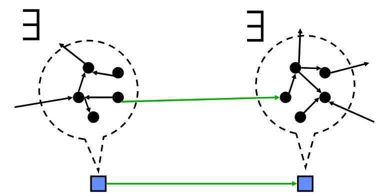
Abstract Kripke Structure

- Abstract interpretation of atomic propositions
  - $I'(a, p)$ = true iff forall s in $\gamma(a)$, $I(s, p)$ = true
  - $I'(a, p)$ = false iff forall s in $\gamma(a)$, $I(s, p)$ = 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.

49

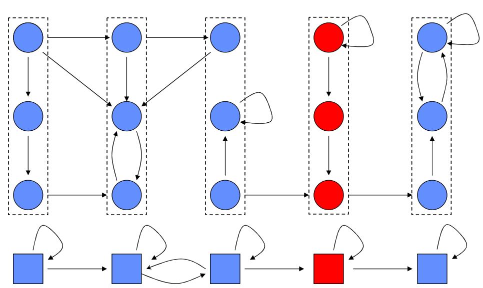
Computing Over-Approximation

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



50

Over-Approximation (Existential Abstraction)



51

Preservation via Over-Approximation

Let φ be a universal temporal formula (ACTL, LTL)

Let K' be an over-approximating abstraction of K

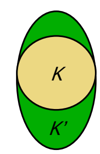
Preservation Theorem

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

Converse does not hold

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

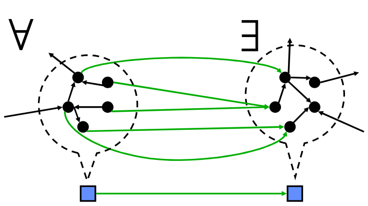
K' may have extra behaviors



52

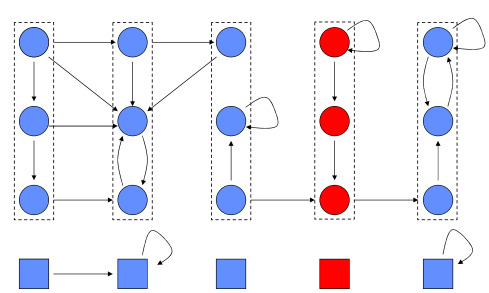
Computing Under-Approximation

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



53

Under-Approximation (Universal Abstraction)



54

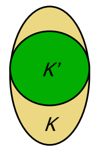
Preservation via Under-Approximation

Let φ 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

Which abstraction to use?

Property Type	Expected Result	Abstraction to use
Universal (ACTL, LTL)	True	Over-
	False	Under-
Existential (ECTL)	True	Under-
	False	Over-

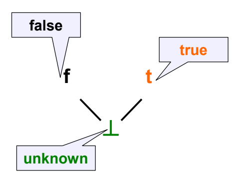
But what about mixed properties?!

Part II: Abstraction

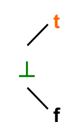
- Defining an Abstract Domain
  - variable elimination, data abstraction, predicate abstraction
- Abstraction for Universal/Existential Properties
  - over- and under-approximations
- Abstraction for Mixed Properties
  - 3-valued abstraction
- Overlapping Abstract Domains
  - Belnap (4-valued) abstraction

3-Valued Kleene Logic

Information Ordering



Truth Ordering



$t \wedge \perp = \perp$

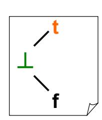
$t \vee \perp = t$

$\neg t = f$

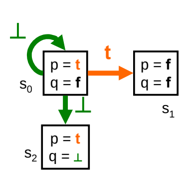
$\neg \perp = \perp$

3-Valued Kripke Structures

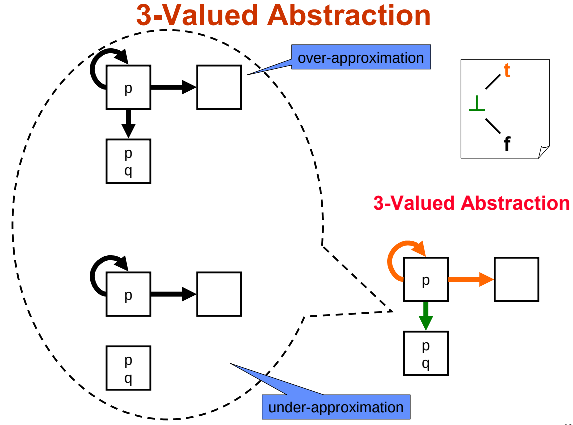
- Kripke structures extended to 3-valued logic



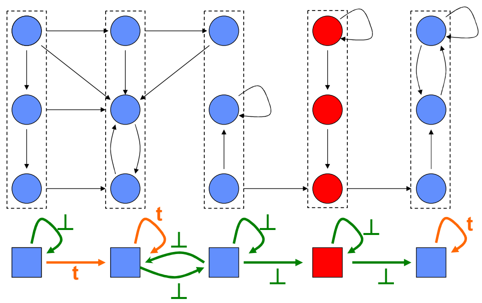
- Propositions can be
  - True, False, or Unknown
- Transitions
  - possible: $\perp$
  - necessary and possible: t
  - impossible: f



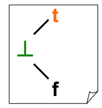
3-Valued Abstraction



Example Revisited (3-Val Abstraction)

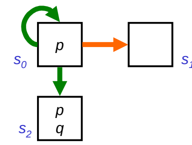


Model-Checking with 3 Values



- Usual semantics of temporal operators
- BUT connectives $\wedge \vee \neg$ are interpreted in 3-Valued Logic

$(EX\ \varphi)(s) = \bigvee_t R(s,t) \wedge \varphi(t)$



Examples

$(EX\ \neg p)(s_0) = \mathbf{t}$

$(EX\ q)(s_0) = \perp$

$(EX\ \neg p \wedge q)(s_0) = \mathbf{f}$

Preservation via 3-Valued Abstraction

Let φ be a temporal formula (CTL)

Let K' be a 3-valued abstraction of K

Preservation Theorem

Abstract MC Result	Concrete Information
True (t)	$K \models \varphi$
False (f)	$K \models \neg\varphi$
Maybe ($\perp$)	$K \models \varphi$ or $K \models \neg\varphi$

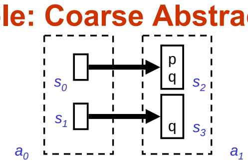
no information

Preserves truth and falsity of arbitrary properties!

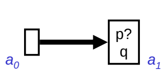
Part II: Abstraction

- **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

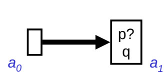


Over-Approximation



AX (p ∨ ¬p) is inconclusive

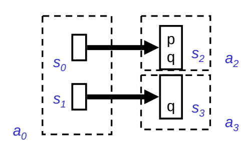
Under-Approximation



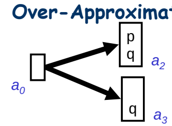
EX (q) is true

Goal: make AX conclusive as well, via domain refinement

Example: Refined Abstract Domain

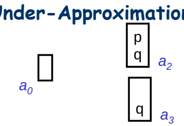


Over-Approximation



AX (p ∨ ¬p) is true

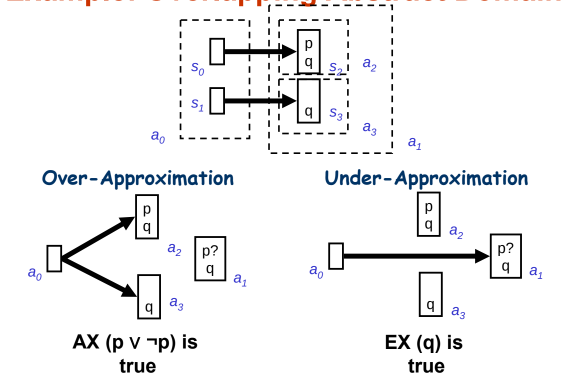
Under-Approximation



EX (q) is inconclusive

Partitioned domain does not work!
Need an overlapping abstract domain!!!

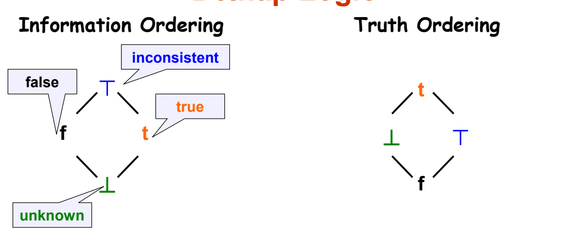
Example: Overlapping Abstract Domain



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



$t \wedge \bot = \bot$

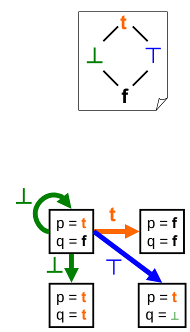
$t \vee \bot = t$

$\neg t = f$

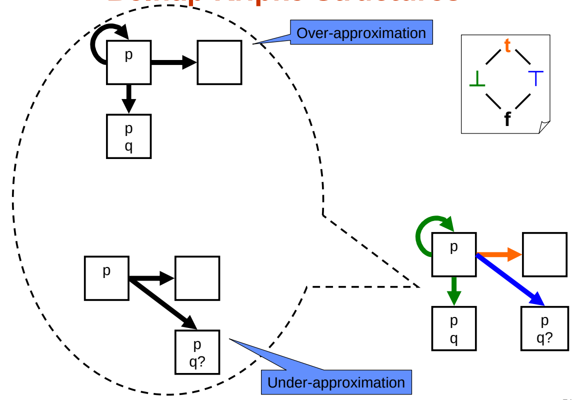
$\neg \bot = \bot$

Belnap Kripke Structures

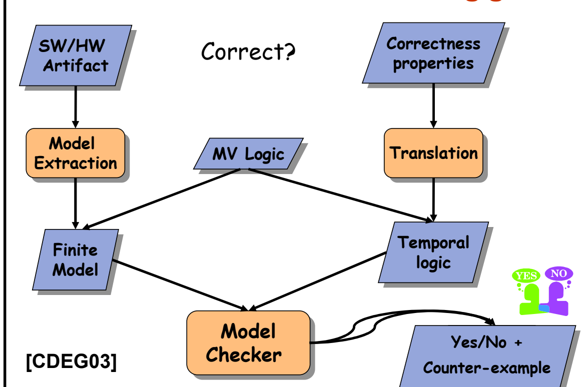
- Kripke structures extended to Belnap logic
- Propositions
  - True, False, or Unknown
- Transitions
  - only under-approximation: ⊤
  - only over-approximation: ⊥
  - both over- and under-: t
  - neither: f



Belnap Kripke Structures



Multi-Valued Model Checking



[CDEG03]

Preservation via Belnap Abstraction

Let φ be a temporal formula (CTL)

Let K' be a Belnap abstraction of K

Preservation Theorem

Abstract MC Result	Concrete Information
True	$K \models \varphi$
False	$K \models \neg\varphi$
$\perp$	$K \models \varphi$ or $K \models \neg\varphi$
$\top$	$K \models \varphi$ and $K \models \neg\varphi$

Not possible for a sound abstraction

Preserves truth and falsity of arbitrary properties!

Summary

Abstraction is the key to scaling up

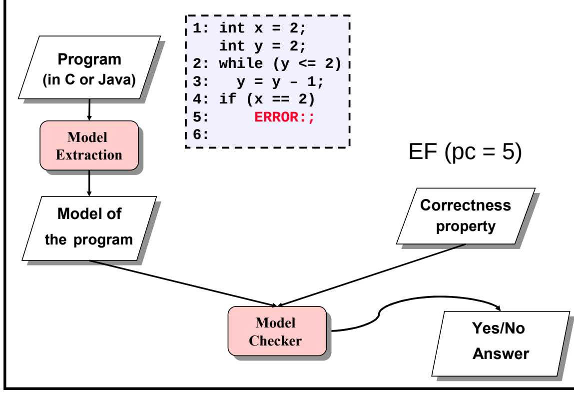
1. Choose an abstract domain
   - Variable elimination, data abstraction, predicate abstraction, …
2. Choose a type of abstraction
   - Over-, Under-, 3Val, Belnap
3. Build an abstract model ($$$$$)
4. Model-check the property on the abstract model
5. If the result is conclusive, STOP
6. Otherwise, pick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

Next: Software Model Checking and Abstraction

Part III

Software Model Checking

Software Model Checking



In Our Programming Language…

- All variables are global
- Functions are inlined
- `int` is integer
  - i.e., no overflow
- Special statements:

`skip`	do nothing
`assume(e)`	if `e` then `skip` else abort
`x,y=e1,e2`	`x`, `y` are assigned `e1`,`e2` in parallel
`x=nondet()`	`x` gets an arbitrary value
`goto L1,L2`	non-deterministically go to `L1` or `L2`

From Programs to Kripke Structures

Program

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

State

pc	x	y	…
3	1	3	…

Step

pc	x	y	…
2	1	2	…

Property: EF (pc = 5)

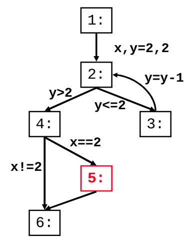
Programs as Control Flow Graphs

Program

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

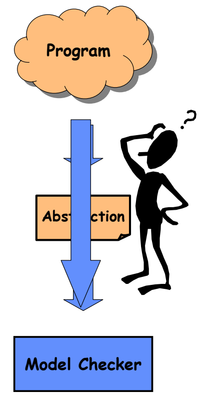
Semantics S

Labeled CFG



79

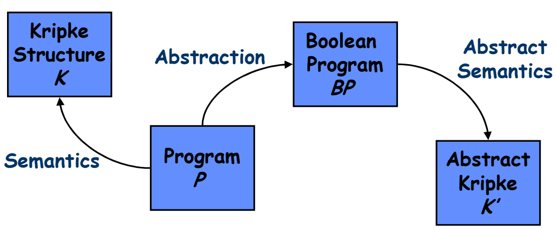
Model Checking Software



- 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

80

Software Model Checking and Abstraction



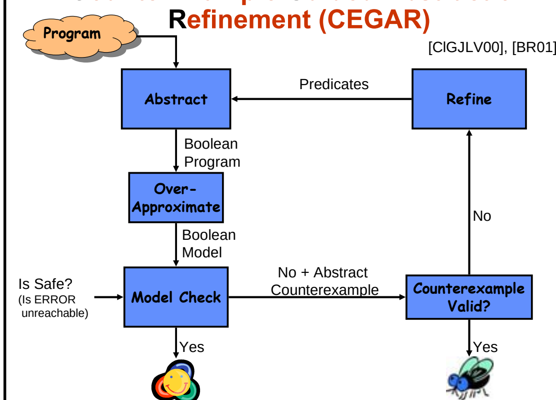
Soundness of Abstraction:

BP abstracts *P* implies that *K'* approximates *K*

81

CounterExample Guided Abstraction Refinement (CEGAR)

[ClGJLV00], [BR01]



82

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

83

The Running Example

Program	Property	Expected Answer

`1: int x = 2;`
`   int y = 2;`
`2: while (y <= 2)`
`3:    y = y - 1;`
`4: if (x == 2)`
`5:      ERROR:;`
`6:`
	EF (pc = 5)	False

84

An Example Abstraction

Program

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

Abstraction

(with y<=2)

```
bool b is (y <= 2)
1: b = T;

2: while (b)
3:   b = ch(b,f);
4: if (*)
5:     ERROR:;
6:
```

85

Boolean (Predicate) Programs (BP)

- Variables correspond to predicates
- Usual control flow statements
  - while, if-then-else, goto
- Expressions
  - usual Boolean expressions

`*` — unknown

`ch(a,b)` — if a then true else if b then false else *

- Parallel Assignment
  - p_1 = ch(a_1,b_1), p_2 = ch(a_2,b_2), ...
  - b_1 = ch(b_1,¬b_1), b_2 = ch($b_1 \vee b_2$, f), b_3=ch(f,f)

86

Detour: Weakest Preconditions

Def. A *Ho* ... ds
when
For an
ter

{3>y} x = 3 {x>y} ✓

{x>0} x = 2+y {y>0} ✗

{*x>3 ∨ x = &y} y=5 {*x>3} ✓

{false} y=5 {y<0} ✓

Def. The
(denoted WP(C,Q)) is a formula P such that
1. {P} C {Q}
2. for all other P′ such that {P′} C {Q}, P′ ⇒ P (P is weaker then P′).

87

Calculating Weakest Preconditions

- Assignment (easy)
  - WP (x=e, Q) = Q[x/e]
    - Intuition: after an assignment, 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==y)[x/0] = (0==y)
  - WP (x=0, x==y+1) = (x==y+1)[x/0] = (0 == y+1)
  - WP (y=y-1,y<=2) = (y<=2)[y/y-1] = (y-1 <= 2)
  - WP(y=y-1,x==2) = (x==2)[y/y-1] = (x == 2)

88

Boolean Program Abstraction

- Update p = ch(a, b) is an approximation of a concrete statement S iff {a}S{p} and {b}S{¬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

89

Computing An Abstract Update

```
// S a statement under abstraction
// P a list of predicates used for abstraction
// t a target predicate for the update
absUpdate (Statement S, List<Predicates> P, Predicate q) {
  resT, resF = false, false;
  // foreach full conjunction of literals in P
  foreach m : monomials(P) {
    if (tpQ("m ⇒ WP(S,q)") resT = resT ∨ m;
    if (tpQ("m ⇒ WP(S,¬q)") resF = resF ∨ m;
  }
  return "q = ch(resT, resF)"
}
```

90

absUpdate (y=y-1, P={y<=2}, q=(y<=2))

```
y = y - 1;
```

absUpdate

```
(y<=2) = ch (y<=2,f)
```

```
P is  {y <= 2}
q is  (y <= 2)
```

```
WP(y=y-1,y<=2)      is (y -1) <= 2
WP(y=y-1,¬(y<=2)) is (y – 1) > 2
```

Theorem Prover Queries:

```
(y<=2) ⇒ (y – 1) <= 2   ✓
¬(y<=2)⇒ (y – 1) <= 2   ✗
(y<=2) ⇒ (y – 1) >  2   ✗
¬(y<=2)⇒ (y – 1) >  2   ✗
```


The result of abstraction

Program

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

Abstraction
(with y<=2)

```
bool b is (y <= 2)
1: b = T;

2: while (b)
3:    b = ch(b,f);
4: if (*)
5:     ERROR:;
6:
```

But what is the semantics of Boolean programs?

BP Semantics: Overview

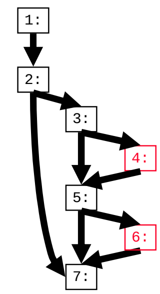
- Over Approximation
  - treat "unknown" as non-deterministic
  - good for establishing correctness of universal properties
- Under Approximation
  - treat "unknown" as abort
  - good for establishing failure of universal properties
- Exact Approximation
  - Treat "unknown" as a special unknown value
  - good for verification and refutation
  - good for universal, existential, and mixed properties

BP Semantics: Over-Approximation

Abstraction

```
1: ;
2: if (nondet) {
3:   if (*)
4:      ERROR;
5:   if (nondet)
6:     ERROR;
7: }
```

Over-Approximation



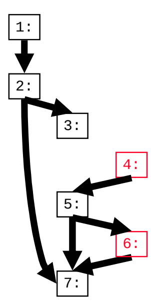
Unknown is treated as non-deterministic

BP Semantics: Under-Approximation

Abstraction

```
1: ;
2: if (nondet) {
3:   if (*)
4:      ERROR;
5:   if (nondet)
6:     ERROR;
7: }
```

Under-Approximation



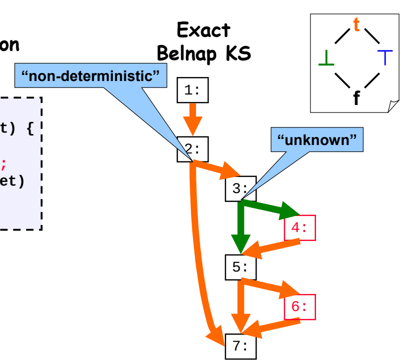
Unknown is treated as abort

BP Semantics: Exact Approximation

Abstraction

```
1: ;
2: if (nondet) {
3:   if (*)
4:      ERROR;
5:   if (nondet)
6:     ERROR;
7: }
```

Exact Belnap KS



Unknown is treated as unknown

Summary: The Three Semantics

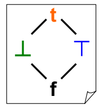
Concrete

```
y = y - 1;
```

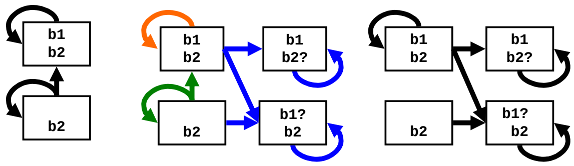
Abstract

b1 is (y <= 2)
b2 is (x == 2)

```
b1 = ch(b1,f);
b2 = ch(b2,¬b2)
```

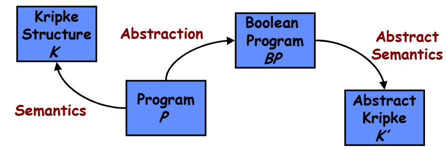


Over-Approx | **Belnap (Exact)** | **Under-Approx**



97

Summary: Program Abstraction



1. Abstract a program *P* by a boolean program *BP*
2. Pick an abstract semantics for this *BP*:
   1. **Over-approximating**
   2. **Under-approximating**
   3. **Belnap (Exact)**
3. Yield relationship between *K* and *K'*:
   1. **Over-approximation (in the sense of Part II)**
   2. **Under-approximation (in the sense of Part II)**
   3. **Belnap abstraction (in the sense of Part II)**

98

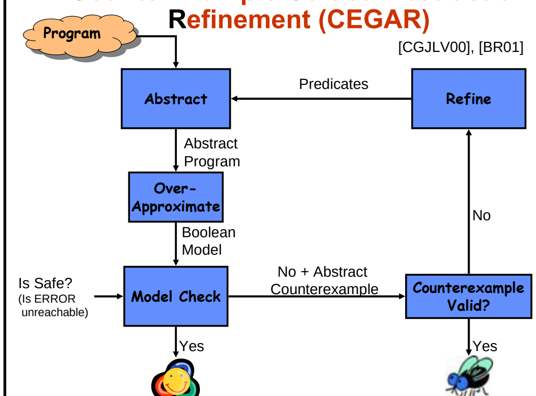
Outline

- 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

99

CounterExample Guided Abstraction Refinement (CEGAR)

[CGJLV00], [BR01]



100

Example: Is ERROR Unreachable?

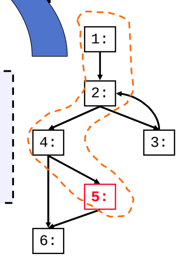
Program | **Abstraction** | **Over-Approximation**

Need This!

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

```
1: ;

2: while (*)
3:      ;
4: if (*)
5:      ERROR:;
6:
```



CEGAR steps

Abstract → Translate → Check → Validate → Repeat

101

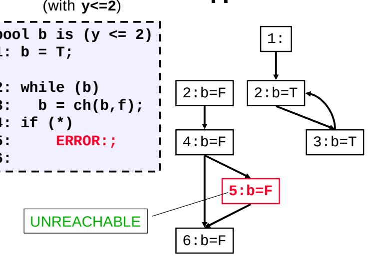
Example: Is ERROR Unreachable?

Program | **Abstraction** (with y<=2) | **Over-Approximation**

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

```
bool b is (y <= 2)
1: b = T;

2: while (b)
3:     b = ch(b,f);
4: if (*)
5:      ERROR:;
6:
```

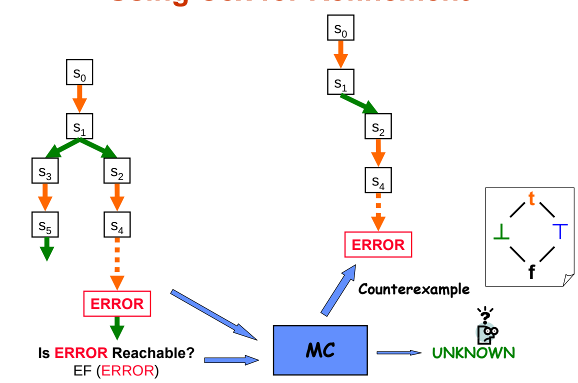


CEGAR steps

Abstract → Translate → Check → NO ERROR

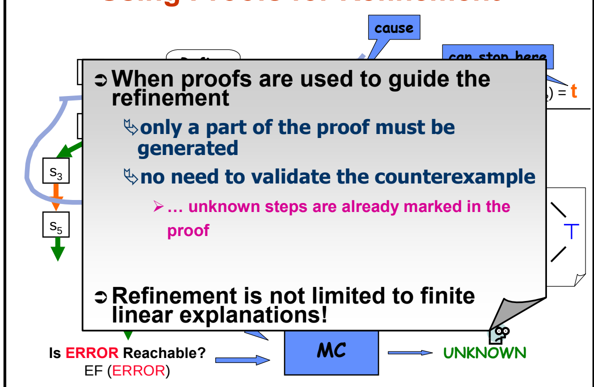
102

Using Cex for Refinement



103

Using Proofs 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!

104

Finding Refinement Predicates

- Recall
  - each abstract state is a conjunction of predicates
    - i.e., y<=2∧x==2 y>2 ∧ x!=2 etc.
  - each abstract transition corresponds to a program statement

Result from a partial proof	MC needs to know validity of
Unknown transition $s_1 \rightarrow s_2$	$\{s_1\}$ C $\{s_2\}$

C is the statement corresponding to the transition

105

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
    - i.e., $\{s_1\}$C{p} is not valid
  - refine by adding WP(C,p)

106

An Example

$s_1 \rightarrow s_2$ is unknown

pc=2 y>2 x==2	→	pc=3 y>2 x==2

```
{y>2∧x==2} y = y-1 {y>2∧x==2}
```

```
{y>2∧x==2} y = y-1 {x==2}
```
✓

new predicate `WP(y = y-1, y>2) = y>3`

107

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.
    - over-, under-, Belnap
- Automatic abstraction refinement enables to find the "right" abstraction incrementally

108

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

109

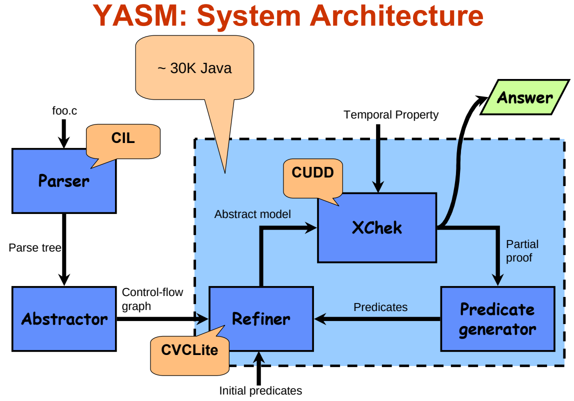
YASM

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

Yet **A**nother **S**oftware **M**odel-checker

110

YASM: System Architecture



111

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

NEW!

112

Current Applications

- BLAST Benchmarks [GC06]
  - Device drivers (4K-6K LOC)
  - Parts of OpenSSH (2K-3K LOC)
- Split OpenSSH (100K LOC)
  - with UofT Security Group
- Detecting setuid/seteuid security flaws
  - with UofT Security Group, in progress
- 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

NEW! NEW!

113

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

114

BLAST

- http://embedded.eecs.berkeley.edu/blast/
- Properties: Reachability
- Abstraction: Predicate over-approximation
- MC Engine: Symbolic BDD-based
  - MC and abstraction are interleaved
- Refinement: Predicates from a proof of impossibility of a counter-example

115

SATABS & CBMC

- http://www.inf.ethz.ch/personal/daniekro/satabs/
- 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

116

MAGIC

- http://www.cs.cmu.edu/~chaki/magic/
- Properties: Automata Simulation
- Abstraction: Predicate over-approximation
- MC Engine: SAT-based
- Refinement: Symbolic simulation of cex
- Key Features: support for concurrent C modules

117

Java PathFinder

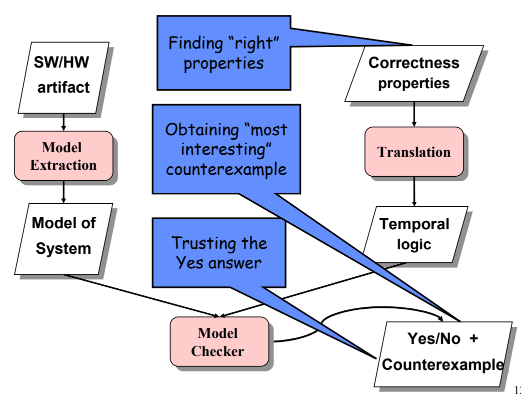
- http://javapathfinder.sourceforge.net/
- Properties: Reachability
- Abstraction: user-provided data abstraction
- MC Engine: Explicit state with symbolic execution
- Refinement: None
- Key Features: support for Java including Objects and Threads

118

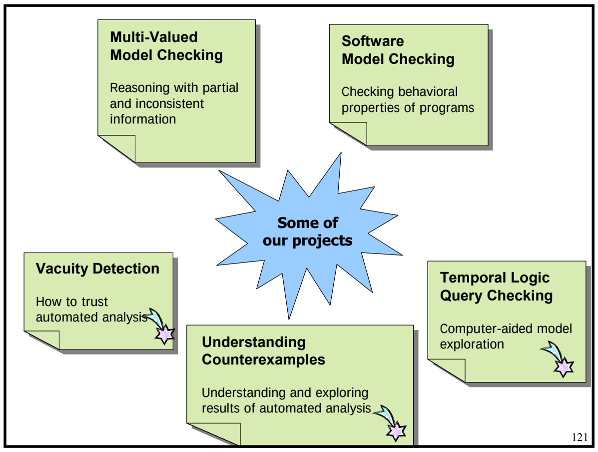
Part IV

Usability Issues

Usability Issues (Our Work)



120



121

Problem of Vacuity

- Computed counterexample – real error
- No such information is given when property holds!
  - e.g., "every request is eventually acknowledged"
    - AG (req ⇒ AF ack)
  - Let *K* be a model that does not produce any requests
    - The formula holds in *K* "too easily" – vacuously
    - Satisfaction does not depend on the model -> modeling error?
- Vacuity – real problem in practice
  - Case studies [BBER01] :
    - 20% of properties hold vacuously
    - Vacuity always points to a problem in the design, its specification, or its environment

122

Dealing with Vacuity: Manual Approach

- Check that antecedent of implication is satisfied in at least one state
  - EF (req) ∧ AG (req ⇒ AF ack)
- Often hard to get right for long properties
- Defeats the purpose of model-checking as automatic technique

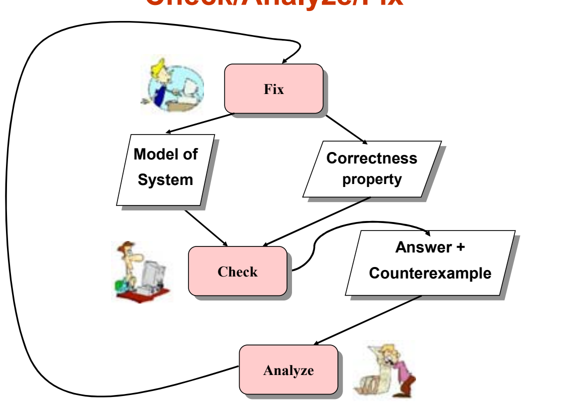
123

Our Vacuity Project [GC04]

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

124

Check/Analyze/Fix



125

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* □*g* fails because BOTH are false

126

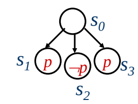
Interactive Explanations

- 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 → 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*) ∨ (EG *q*)
  - **implicit (via EX)**
    - which state to pick as a witness?
    - Example: EX *p*

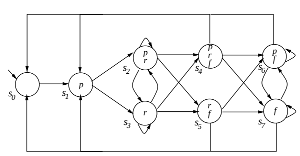


- By default, choice is random, with goal to find shortest witness

Elevator Controller System

Button model

r – request to move has been generated
f – request is fulfilled, button can be reset
(request cannot be fulfilled before generated)
p – state of button (pressed or released)



Task 1: Getting the property right

- **Attempt 1: AG AF (floor = 3 ∧ door = open)**
  - No: can get stuck on first floor
- **Attempt 2: AG(floor≠1 →AF(floor=3 ∧ 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* →AF(floor=3 ∧ 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**

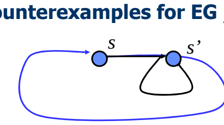
Task 2: Reducing cognitive overload

- 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

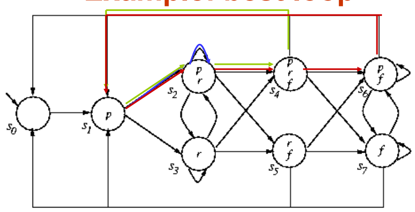
Task 3: choosing "best" loop

- Why? (Attempt at "shortest" counterexample)
  - **Counterexamples for EG *p* properties:**



- Goal: find "best" loop
  - **... around most familiar state (Idle)**
  - **... most interesting, using loop summaries**

Example: best loop



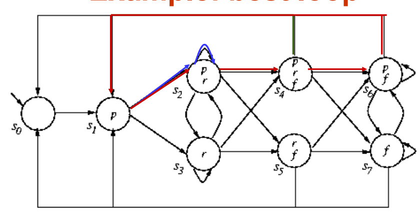
Summaries

$\boxed{?}$ 3-state p loop ∨ $\boxed{?}$ EX EG p — $\boxed{?}$ EG p — s_1

$\boxed{?}$ 1-state p loop ∨ $\boxed{?}$ EX EG p — $\boxed{?}$ EG p — s_2

133

Example: best loop



Strategy:

- **Examine model from s_1 to depth 3 for witnesses to EG p**
- **Choose the shortest loop to explore**

$\boxed{?}$ 3-state p loop ∨ $\boxed{?}$ EX EG p — $\boxed{?}$ EG p — s_1

134

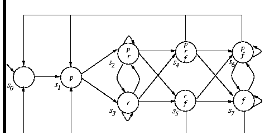
How do we do it?

- **Create: a counter example 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

135

Proofs

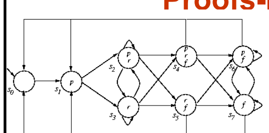


$$\dfrac{\dfrac{\dfrac{\dfrac{R(s_0, s_1) \qquad \dfrac{R(s_1, s_2) \qquad \dfrac{r(s_2)}{(f \vee r)(s_2)}}{EX(f \vee r)(s_1)}}{EX\,EX\,(f \vee r)(s_0)}}{EF_2\,(f \vee r)(s_0)}}{\exists n \in N \cdot\ EF_n\,(f \vee r)(s_0)}}{EF\,(f \vee r)(s_0)}$$

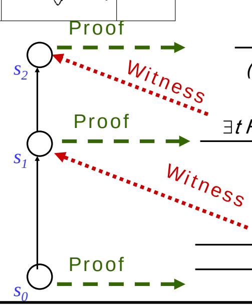
- **Good:**
  - **complete**
    - available for all temporal properties
    - all information is here
- **Bad:**
  - **too verbose**
  - **not particularly intuitive. Where is the model?**

136

Proofs-like Witness



Why does EF $(f \vee r)(s_0)$ hold, i.e., why is $(f \vee r)$ reachable?



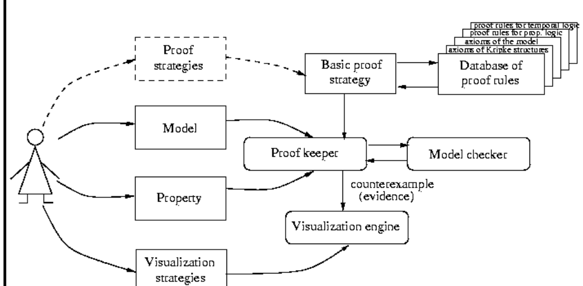
$$\dfrac{r(s_2)}{(f \vee r)(s_2)}$$

$$\dfrac{\exists t\ R(s_1, t) \wedge (f \vee r)(t)}{EX(f \vee r)(s_1)}$$

$$\dfrac{\dfrac{\overline{EX\,EX\,(f \vee r)(s_0)}}{EF_2\,(f \vee r)(s_0)}}{EF\,(f \vee r)(s_0)}$$

137

Overall Framework



138

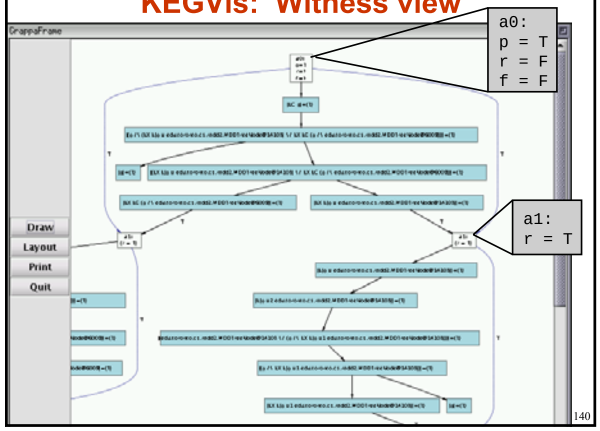
Visualization Engine

- Produce proof-like counterexamples
  - And present parts of model
- Present proof summaries ("what is going to follow")

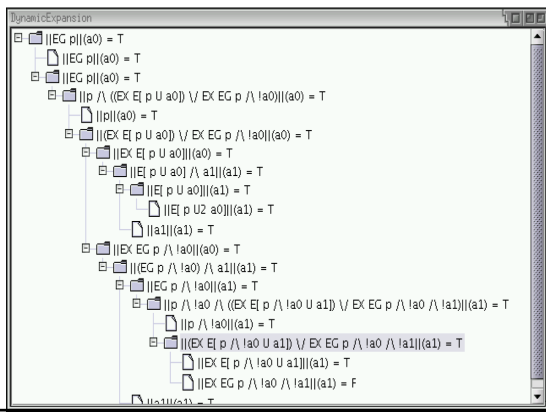
➲ **Visualization strategies**

- Restrict scope of explanation (starting/stopping)
  - <u>Example</u>: EG EF ($x \wedge$ EX x) and want to see witness to EF
  - Starting condition: EF ($x \wedge$ EX x)
  - Stopping condition: $x \wedge$ 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



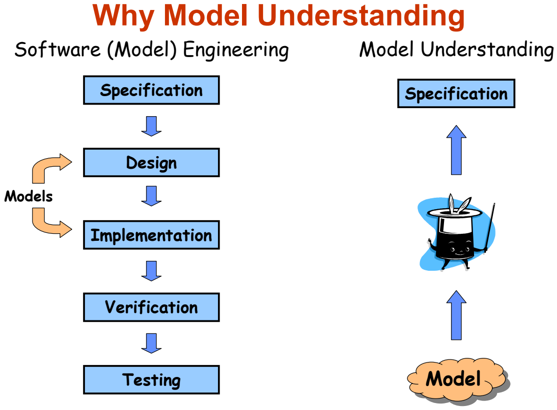
KEGVis: Proof View



Temporal Logic Query Checking

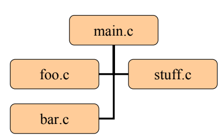
Computer-Aided Model Understanding

Why Model Understanding

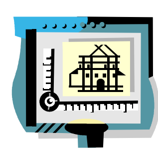


Model Understanding - Structural

➲ **Modules and dependencies**

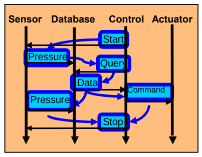


➲ **Design and architectural patterns**



Model Understanding - Behavioural

- Scenarios
  - sample behaviors

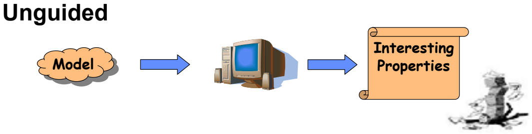


- Properties
  - succinct summaries of behaviors

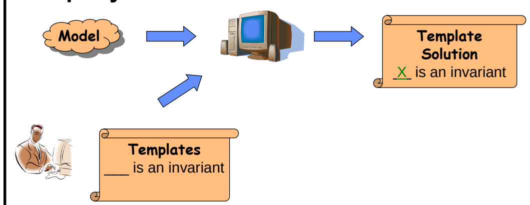
"X is an invariant"
"X is eventually followed by Y"

Computer-Aided Model Understanding

- Unguided



- Property Guided



Example: Cruise Control System (CCS)

- Maintains a speed of an automobile
  - Four major modes of operation (indicated by variable CC)

CC = Off	cruise control is off
CC = Inactive	cruise control is idle
CC = Cruise	maintaining the speed of the automobile
CC = 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

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?

Query Checking

[Ch00]

TL Property

p is an invariant

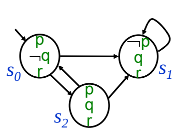
AG p

TL Query

What is an invariant?

AG ?

placeholder



Propositional Solutions

p ∨ q r

(p ∨ q) ∧ r

strongest

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

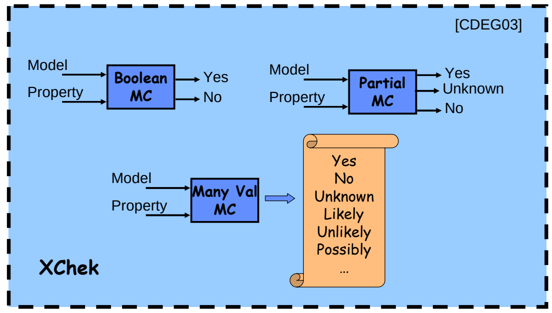
The Language of Queries

➲ Queries based on arbitrary CTL properties, with

- **multiple occurrence of a placeholder**
  e.g. "what happens twice in a row?"
  EF (? ∧ EX ?)

- **several different placeholders**
  e.g. "what states can follow each other?"
  EF (? ∧ EX ?)
- **allow restrictions on placeholders**
  e.g. "what modes can follow each other?"
  EF (?{CC} ∧ EX ?{CC})

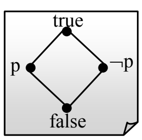
Detour: Multi-Valued Model-Checking



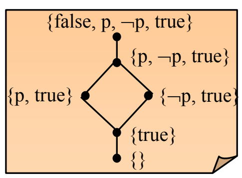
➲ when values form a lattice

- .. theory and implementation of all these tools is the same!

Query-Checking as Multi-Valued MC



Lattice of propositional formulas over {p} on implication

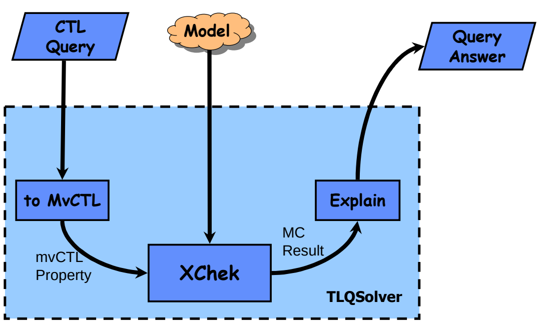


Lattice of possible solutions on set inclusion

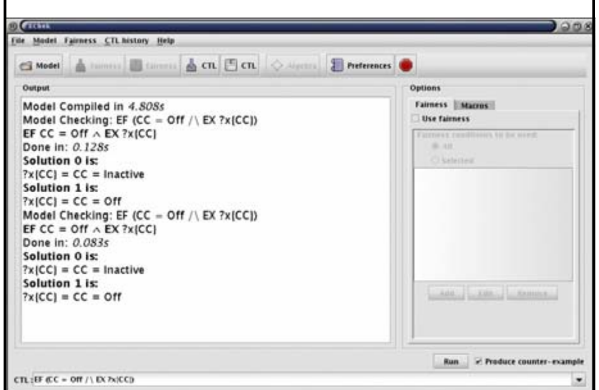
- **Multi-valued model-checker can iterate over such lattices**
  - ... giving the appropriate element as a solution
- **Reduce query-checking to multi-valued model-checking!**

TLQSolver

Query-Checking via Multi-Valued Model-Checking



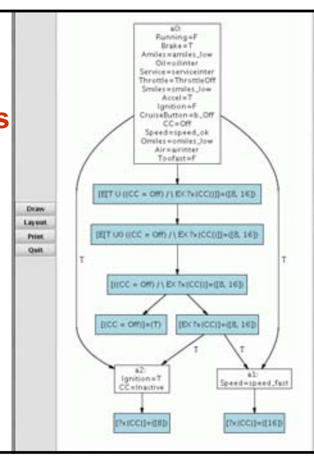
TLQSolver



Generated Witness

Query:

EF((CC=Off) ∧ *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!**

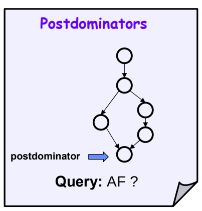
- Guided simulation
  - specify ***objective*** via a query
  - witness serves as basis for simulation
  - Example:
    - goal: *EF* ? {CC}
    - prefer witnesses with largest common prefix

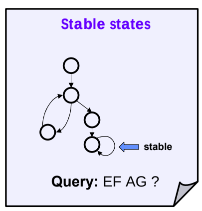
Other Applications

- Invariant discovery
  - e.g., what is true when CCS is in mode Cruise
- Pre-condition 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":





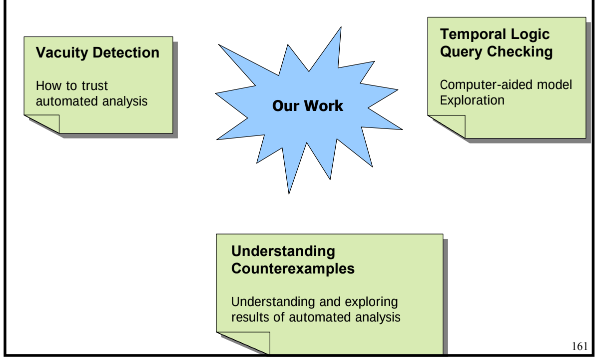
- State-based queries can be solved efficiently
  - polynomial in # of states of the model
    - See FM '06 demo!

NEW!

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



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

[BBER01] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. "Efficient Detection of Vacuity in Temporal Model Checking". FMSD, Vol. 18, No. 2, pp. 141-163, March 2001.

[BR01] T. Ball, S. Rajamani. "The SLAM Toolkit". In CAV'01.

[Ch00] W. Chan. "Temporal Logic Queries". In CAV'00.

[CDEG03] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel, "Multi-Valued Symbolic Model-Checking". In TOSEM, No. 4, Vol. 12, pp. 1-38, 2003.

[CG06] M. Chechik, A. Gurfinkel. "A Framework for Counterexample Generation Exploration", to appear in *STTT'06* (shorter versions in TACAS'03 and FASE'05).

[CGJLV00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. "Counterexample-Guided Abstraction Refinement". In CAV'00.

[DGG97] D. Dams, R. Gerth, and O. Grumberg, "Abstract Interpretation of Reactive Systems". In TOPLAS, No. 19, Vol. 2, pp. 253-291, 1997.

163

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.

[GCD03] A. Gurfinkel, M. Chechik, B. Devereux. "Temporal Logic Query Checking: A Tool for Model Exploration". *IEEE Transactions on Software Engineering*, Vol. 29, No. 10, pp. 898-914, October 2003 (shorter versions in FSE'02 and CAV'03).

[Mc93] K. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[PPV05] C. Pasareanu, R. Pelanek, W. Visser. "Concrete Search with Abstract Matching and Refinement". In CAV'05.

164

Acknowledgements

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

165