Model Checking, Temporal Logic, and Automata Theory

Bow-Yaw Wang

Institute of Information Science
Academia Sinica, Taiwan

July 12, 2007

Introduction

- Consider the Needham-Schröder Authentication Protocol

$$
\begin{array}{ccc}
A & \xrightarrow{\{A,N_A\}_{K_B}} & B \\
A & \xleftarrow{\{N_A,N_B\}_{K_A}} & B \\
A & \xrightarrow{\{N_B\}_{K_B}} & B
\end{array}
$$

- How do you know the protocol is "correct?"
  - Prove it by hand
    - sometimes it is very tedious and thus error-prone
  - Verify it by machine
    - Let's do it!

Introduction (cont'd)

- Here is the buggy trace found by OMOCHA

$$
\begin{array}{ccccc}
A & \xrightarrow{\{A,N_A\}_{K_E}} & E & & \\
 & & E & \xrightarrow{\{A,N_A\}_{K_B}} & B \\
A & \xleftarrow{\{N_A,N_B\}_{K_A}} & E & \xleftarrow{\{N_A,N_B\}_{K_A}} & B \\
A & \xrightarrow{\{N_B\}_{K_E}} & E & & \\
 & & E & \xrightarrow{\{N_B\}_{K_B}} & B
\end{array}
$$

Introduction (cont'd)

- Generally, it is undecidable to verify an arbitray property on an arbitrary algorithm
- Model checking hence focuses on verifying limited classes of properties on restricted computation models
  - linear temporal logic, computational tree logic, μ-calculus
  - finite-state automata, pushdown automata, Petri nets
- In this lecture, we will discuss automatic verification of linear temporal logic and computational tree logic on finite-state automata
- Moreover, you will have a chance to use tools to verify some protocols automatically

Relation to Other Topics

- from logic to temporal logic
- "realistic" functional programming
- applying type system

Outline

Kripke Structures

- Let AP be the set of *atomic propositions*. A *Kripke structure* $K = (Q, Q_0, \delta, L)$ is a triple where
  - Q is a set of *states*;
  - $Q_0 \subseteq Q$ is the set of *initial states*;
  - $\delta \subseteq Q \times Q$ is a (total) *transition relation*;
  - $L : Q \to 2^{AP}$
- As usual, we write $q \longrightarrow q'$ for $(q, q') \in \delta$
- A *computation path from* q is an infinite sequence of states $\pi = p_0 p_1 \cdots p_n \cdots$ with $p_0 = q$ and $p_i \longrightarrow p_{i+1}$ for $0 \leq i$
- Define $\pi(i) = p_i p_{i+1} \cdots$

Linear Temporal Logic – Syntax

- An atomic proposition is an LTL formula
- If f and g are LTL formulae
  - $\neg f$ and $f \vee g$ are LTL formuae
  - $f\,\mathbf{U}g$ is an LTL formula

Linear Temporal Logic – Semantics

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure.
- Given a computation path $\pi = p_0 p_1 \cdots p_n \cdots$ and an LTL formula f, define the *satisfaction relation* $K, \pi \models f$ by
  - $K, \pi \models ap$ if $ap \in L(p_0)$
  - $K, \pi \models \neg f$ if not $K, \pi \models f$
  - $K, \pi \models f \vee g$ if $K, \pi \models f$ or $K, \pi \models g$
  - $K, \pi \models \mathbf{X} f$ if $K, \pi(1) \models f$
  - $K, \pi \models f \mathbf{U} g$ if there is a $k \geq 0$ such that $K, \pi(k) \models g$ and $K, \pi(j) \models f$ for $0 \leq j < k$
- We will use the following abbreviation

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
f \wedge g & \equiv & \neg(\neg f \vee \neg g) & \mathbf{F} f & \equiv & \mathrm{true} \mathbf{U} f \\
\mathbf{G} f & \equiv & \neg \mathbf{F} \neg f & & &
\end{array}
$$

Linear Temporal Logic – in Plain English

- $K, \pi \models ap$: ap holds initially
- $K, \pi \models \mathbf{X}f$: f holds at next position
- $K, \pi \models f\mathbf{U}g$: f holds until g holds
- $K, \pi \models \mathbf{F}f$: f holds eventually
- $K, \pi \models \mathbf{G}f$: f always holds

Computational Tree Logic – Syntax

- An atomic proposition is a CTL formula
- If f and g are CTL formulae
  - $\neg f$ and $f \vee g$ are CTL formuae
  - $\mathbf{A}(f\mathbf{U}g)$ and $\mathbf{E}(f\mathbf{U}g)$ are CTL formulae

Computational Tree Logic – Semantics

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure.
- Given a state $q \in Q$ and a CTL formula f, define the *satisfaction relation* $K, q \models f$ as follows
  - $K, q \models ap$ if $ap \in L(q)$
  - $K, q \models \neg f$ if not $K, q \models f$
  - $K, q \models f \vee g$ if $K, q \models f$ or $K, q \models g$
  - $K, q \models \mathbf{EX} f$ if $K, q' \models f$ for some q' with $q \to q'$
  - $K, q \models \mathbf{A}(f \mathbf{U} g)$ if $K, \pi \models f \mathbf{U} g$ for all computation path π from q
  - $K, q \models \mathbf{E}(f \mathbf{U} g)$ if $K, \pi \models f \mathbf{U} g$ for some computation path π from q
- We will use the following abbreviation

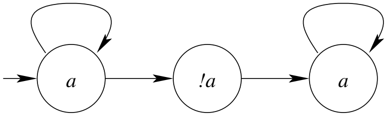
$$
\begin{aligned}
\mathbf{AX} f &\equiv \neg \mathbf{EX} \neg f & \qquad \mathbf{AF} f &\equiv \mathbf{A}(\text{true} \mathbf{U} f) \\
\mathbf{EF} f &\equiv \mathbf{E}(\text{true} \mathbf{U} f) & \qquad \mathbf{AG} f &\equiv \neg \mathbf{EF} \neg f \\
\mathbf{EG} f &\equiv \neg \mathbf{AF} \neg f & &
\end{aligned}
$$

Computational Tree Logic – in Plain English

- $K, q \models ap$: ap holds at q
- $K, q \models \mathbf{AX}f$: f holds at the next position in all paths
- $K, q \models \mathbf{EX}f$: f holds at the next position in some paths
- $K, q \models \mathbf{AF}f$: f holds for all paths from q eventually
- $K, q \models \mathbf{EF}f$: f holds for some path from q eventually
- $K, q \models \mathbf{AG}f$: f always holds for all paths from q
- $K, q \models \mathbf{EG}f$: f always holds for some path from q
- $K, q \models \mathbf{A}(f\mathbf{U}g)$: f holds until g holds for all paths from q
- $K, q \models \mathbf{E}(f\mathbf{U}g)$: f holds until g holds for some path from q

Basic Properties

- LTL and CTL are not comparable
  - There is a property on some Kripke structure, which is expressible by LTL but not CTL and vice versa



- **FG**a holds for all computation paths from q_0
- but **AFAG**a does not hold on q_0

Automata Theory

- Finite State Automata and Regular Languages
  - a simple model accepts finite strings
- ω-Automata and ω-Regular Languages
  - an extension of simple model accepts infinite strings

Strings and Languages

- Consider a set of *alphabets* Σ
- A *string* α is a finite sequence of symbols $a_1 a_2 \cdots a_n$
  - a, abc, $verification$, ...
- The *length* of a string $\alpha = a_1 a_2 \cdots a_n$ is n
- The string of length 0 is called *empty string* and denoted by ϵ
- The set of all strings over Σ is denoted by Σ^*
- A subset of Σ^* is called a *language*

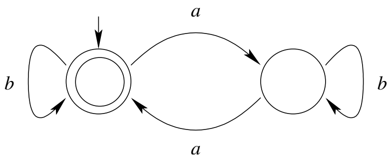
Finite State Automata

- A *finite state automaton* (or *finite state machine*) is a tuple $M = (\Sigma, Q, Q_0, \delta, F)$ where
  - Σ is a finite set of *alphabets*;
  - Q is a finite set of *states*;
  - $Q_0 \subseteq Q$ is the set of *initial states*;
  - $\delta \subseteq Q \times \Sigma \times Q$ is a (total) *transition relation*; and
  - $F \subseteq Q$ is a set of *final states*.
- We will write $q \xrightarrow{a} q'$ for $(q, a, q') \in \delta$
- If $|Q_0| = 1$ and δ is in fact a function from $Q \times \Sigma$ to Q, we say the automaton is *deterministic*

Runs and Acceptance

- Let $M = (\Sigma, Q, Q_0, \delta, F)$ be a finite state automaton and $\alpha = a_1 a_2 \cdots a_n$
- A *run* for α on M is a squence of states $p_0 p_1 \cdots p_n$ such that
  - $p_0 \in Q_0$;
  - $p_i \xrightarrow{a_i} p_{i+1}$ for $0 \leq i < n$
- The set of runs for α on M is denoted by $Run_M(\alpha)$
- The string α is *accepted* by M if there is a run $p_0 p_1 \cdots p_n \in Run_M(\alpha)$ such that $p_n \in F$
- The language accepted by M is denoted by $L(M)$

Example



- $M = (\{a, b\}, \{q_0, q_1\}, \{q_0\}, \delta, \{q_0\})$ where
  - $q_0 \xrightarrow{a} q_1$; $q_0 \xrightarrow{b} q_0$; $q_1 \xrightarrow{a} q_0$; $q_1 \xrightarrow{b} q_1$
- $L(M) = \{\alpha : a \text{ occurs even number of times in } \alpha\}$

Basic Properties

- Nondeterminism does not increase expressiveness
  - A language is accepted by a deterministic finite state automaton if and only if it is accepted by a nondeterministic finite state automaton
- The class of languages accepted by finite state automaton is the class of regular languages
  - RR', $R + R'$, R^*, $\overline{R}$

Infinite Strings and Languages

- An ω-*string* σ is an infinite sequence of symbols $s_1 s_2 \cdots s_n \cdots$
  - $aa \cdots$, $0100011011 \cdots$, ...
- The set of all ω-strings over Σ is denoted by Σ^ω
- A subset of Σ^ω is called an ω-*language*

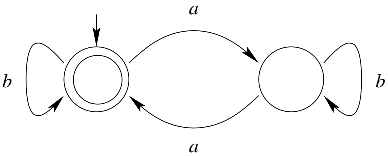
Automata to ω-Automata

- How to make finite state automata accept ω-strings?
- $M = (\Sigma, Q, Q_0, \delta, F)$: a finite state automaton
- A *run* for $\sigma = s_1 s_2 \cdots s_n \cdots$ on M is an infinite sequence of states $p_1 p_2 \cdots p_n \cdots$ such that
  - $p_0 \in Q_0$
  - $p_i \xrightarrow{s_i} p_{i+1}$ for $0 \leq i$
- How to define acceptance?
  - There is no "last" state in an infinite run

Acceptance for ω-Strings

- Let $\sigma = s_1 s_2 \cdots s_n \cdots$ be an ω-string and $B = (\Sigma, Q, Q_0, \delta, F)$ a finite state automaton
- A *run* r for σ on B is an infinite sequence of states $p_1 p_2 \cdots p_n \cdots$ such that
  - $p_0 \in Q_0$
  - $p_i \xrightarrow{s_i} p_{i+1}$ for $0 \leq i$
- Define $\mathit{Inf}_B(r)$ to be the set of states which occur infinitely many times in r
- The *Büchi acceptance condition* for a run r requires $\mathit{Inf}_B(r) \cap F \neq \emptyset$
- An ω-string σ is *accepted* by B if there is an $r \in \mathit{Run}_B(\sigma)$ satisfying the Büchi acceptance condition
- A finite state automaton using the Büchi acceptance condition is called a *Büchi automaton*

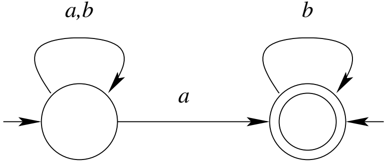
Example



- $bb\cdots$, $aabb\cdots$, $ababb\cdots$, $\ldots$
- $abb\cdots$, $babb\cdots$, $\ldots$
- $L(M) = \{\sigma :$ there are infinitely many or even number of a in $\sigma\}$

Basic Properties

- Deterministic Büchi automata is strictly less expressive than nondeterministic ones
  - The language $\{\sigma : \sigma \text{ contains finitely many } a\text{'s}\}$ is accepted by a nondeterministic Büchi automaton but not by any deterministic Büchi automaton.



Basic Properties (cont'd)

Proof.

Suppose there is a deterministic Büchi automaton $B = (\Sigma, Q, \{q_0\}, \delta, F)$ accepting the same language. Then there is an n_0 such that $q_0 \xrightarrow{b^{n_0}} q$ for some $q \in F$. Otherwise, B would not accept b^ω, a contradiction. Similarly, there must be an n_1 such that $q_0 \xrightarrow{b^{n_1} a b^{n_1}} q$ for some $q \in F$. Hence there are $n_0, n_1, \ldots, n_m, \ldots$ such that

$$q_0 \xrightarrow{b^{n_0} a b^{n_1} \cdots a b^{n_m}} q$$

for some $q \in F$. But the ω-string $b^{n_0} a b^{n_1} \cdots a b^{n_m} \cdots$ contains infinitely many a's. $\square$

Generalized Büchi Automata

- A *generalized Büchi Automaton* $B = (\Sigma, Q, Q_0, \delta, \mathcal{F})$ consists of
  - Σ, a finite set of *alphabets*
  - Q, a finite set of *states*
  - $Q_0 \subseteq Q$, the set of *initial states*
  - $\delta \subseteq Q \times \Sigma \times Q$, the *transition relation*
  - $\mathcal{F} \subseteq 2^Q$, a finite class of *accepting sets*
- The *generalized Büchi acceptance condition* for a run r requires $\mathit{Inf}_B(r) \cap F \neq \emptyset$ for all $F \in \mathcal{F}$
- An ω-string σ is *accepted* by B if there is an $r \in \mathit{Run}_B(\sigma)$ satisfying the generalized Büchi acceptance condition
- A finite state automaton using the generalized Büchi acceptance condition is called a *generalized Büchi automaton*

Expressive Power of Generalized Büchi Automata

- Let $B = (\Sigma, Q, Q_0, \delta, F)$ be a Büchi automaton. It is easy to see that $G_B = (\Sigma, Q, Q_0, \delta, \{F\})$ is a generalized Büchi automaton accepting the same language
- Conversely, let $G = (\Sigma, Q, Q_0, \delta, \mathcal{F})$ be a generalized Büchi automaton with $\mathcal{F} = \{F_0, F_1, \ldots, F_{n-1}\}$
- Construct $B_G = (\Sigma, Q \times \mathbb{N}, Q_0 \times \{0\}, \delta', Q \times \{0\})$ as follows
  - $((q, m), s, (q', m')) \in \delta'$ if and only if
    - $(q, s, q') \in \delta$
    - $m' = \begin{cases} m & \text{if } q' \notin F_m \\ m + 1 \bmod n & \text{if } q' \in F_m \end{cases}$
- Intuitively, we iterate through all accepting sets by a counter
- A run visits all accepting sets infinitely many times if and only if the counter resets to 0 infinitely many times
- Büchi automata have the same expressive power as generalized Büchi automata

LTL Model Checking Problem

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure and f an LTL formula. We write $K \models f$ if $K, \pi \models f$ for all computation paths π from a state in Q_0
- Given a Kripke structure $K = (Q, Q_0, \delta, L)$ and an LTL formula f, the *LTL model checking problem* is to decide whether $K \models f$

Automata-Theoretic Approach

- The idea is to reduce the LTL model checking problem to the language containment problem in automata theory
- Intuitively, we will
  - translate any Kripke structure to an automaton;
  - translate any LTL formula to another automaton;
  - check whether the language accepted by the former automaton is contained in the language accepted by the latter

From Kripke Structures to Büchi Automata

- Consider any Kripke structure $K = (Q, Q_0, \delta, L)$
- Let $\Sigma_{AP} = 2^{AP}$
- We will construct a Büchi automaton accepting a ω-language in Σ_{AP}^{ω}
- Define $B_K = (\Sigma_{AP}, Q \cup \{\iota\}, \{\iota\}, \delta', Q)$
  - ι is a new state not in Q
  - $(q, s, q') \in \delta'$ if $s = L(q')$ and $(q, s, q') \in \delta$
  - $(\iota, s_0, q_0) \in \delta'$ if $s_0 = L(q_0)$ and $q_0 \in Q_0$
- The alphabets are in fact the set of atomic propositions satisfied in the target state
- Any computation path in K corresponds to an ω-string over Σ_{AP} accepted by B_K and vice versa. Precisely,

$$L(B_K) = \{L(\pi) : \pi \text{ is a computation path from } q_0 \in Q_0\}.$$

From LTL to Büchi Automata

- For any LTL formula f, we would like to construct a Büchi automata B_f over Σ_{AP} accepting all ω-strings satisfying f
- Hence to check whether $K, \pi \models f$ for all π from some state in Q_0 is equivalent to checking $L(B_K) \subseteq L(B_f)$

Fischer-Ladner Closure

- Let f be an LTL formula. The *Fischer-Ladner closure* $C(f)$ is defined as follows (we identify $\neg\neg f'$ with f').

$$C(f) = \{f', \neg f' : f' \text{ is a subformula of } f\}$$

- For example,

$$C(a\mathbf{U}b) = \{a\mathbf{U}b, \neg(a\mathbf{U}b), a, \neg a, b, \neg b\}$$

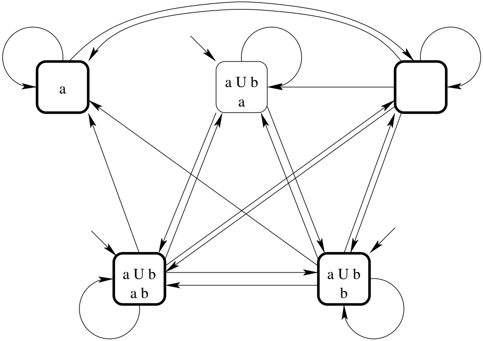
Healthiness

- Let f be an LTL formula. A subset D of $C(f)$ is *healthy* if it satisfies the following conditions
  - for all $f' \in C(f)$, either $f' \in D$ or $\neg f' \in D$;
  - if $f'_0 \vee f'_1 \in C(f)$, then $f'_0 \vee f'_1 \in D$ iff $f'_0 \in D$ or $f'_1 \in D$;
  - if $f'\mathbf{U}g' \in D$, then $g' \in D$ or $f' \in D$;
  - if $f'\mathbf{U}g' \in C(f) \notin D$, then $g' \notin D$.

Automaton $B_f = (\Sigma_{AP}, Q, Q_0, \delta, F)$

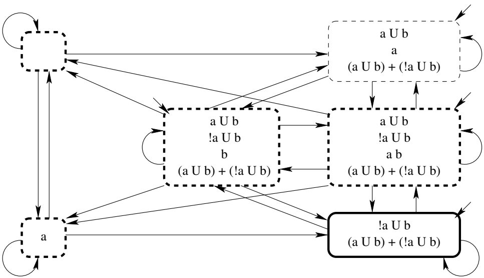
- $Q = \{D : D \text{ is healthy in } C(f)\}$
- $Q_0 = \{D_0 \in Q : f \in D_0\}$
- $(D, s, D') \in \delta$ if
  - $s = D \cap AP$;
  - if $\mathbf{X}f' \in D$, then $f' \in D'$;
  - if $\mathbf{X}f' \in C(f) \notin D$, then $f' \notin D'$;
  - if $f'\mathbf{U}g' \in D$ and $g' \notin D$, then $f'\mathbf{U}g' \in D'$; and
  - if $f'\mathbf{U}g' \in C(f) \notin D$ and $f' \in D$, then $f'\mathbf{U}g' \notin D'$.
- $F = \{F_0, F_1, \ldots, F_n\}$ where $F_i = \{D : f_i'\mathbf{U}g_i' \notin D \text{ or } g_i' \in D\}$ and $f_0'\mathbf{U}g_0'$, $f_1'\mathbf{U}g_1'$, $\ldots, f_n'\mathbf{U}g_n'$ are all subformulae of this form in $C(f)$

$B_{a\mathbf{U}b}$



- $C(a\mathbf{U}b) = \{a\mathbf{U}b, \neg(a\mathbf{U}b), a, \neg a, b, \neg b\}$
- All subsets are $\emptyset$, $\{a\}$, $\{b\}$, $\{a\mathbf{U}b\}$, $\{a, b\}$, $\{a, a\mathbf{U}b\}$, $\{b, a\mathbf{U}b\}$, and $\{a, b, a\mathbf{U}b\}$ All subsets are $\emptyset$, $\{a\}$, $\{b\}$, $\{a\mathbf{U}b\}$, $\{a, b\}$, $\{a, a\mathbf{U}b\}$, $\{b, a\mathbf{U}b\}$, and $\{a, b, a\mathbf{U}b\}$

$B_{(a\mathbf{U}b)\vee(\neg a\mathbf{U}b)}$



- $C((a\mathbf{U}b) \vee (\neg a\mathbf{U}b)) = \{(a\mathbf{U}b) \vee (\neg a\mathbf{U}b), \neg((a\mathbf{U}b) \vee (\neg a\mathbf{U}b)), a\mathbf{U}b, \neg(a\mathbf{U}b), \neg a\mathbf{U}b, \neg(\neg a\mathbf{U}b), a, \neg a, b, \neg b\}$
- Healthy subsets are $\{\}$, $\{a\}$, $\{(a\mathbf{U}b) \vee (\neg a\mathbf{U}b), a\mathbf{U}b, a\}$, $\{(a\mathbf{U}b) \vee (\neg a\mathbf{U}b), \neg a\mathbf{U}b\}$, $\{(a\mathbf{U}b) \vee (\neg a\mathbf{U}b), a\mathbf{U}b, \neg a\mathbf{U}b, b\}$, $\{(a\mathbf{U}b) \vee (\neg a\mathbf{U}b), a\mathbf{U}b, \neg a\mathbf{U}b, a, b\}$
  - Why is $\{(a\mathbf{U}b) \vee (\neg a\mathbf{U}b), a\mathbf{U}b, b\}$ not healty?

Checking Language Containment

- For any LTL formula f, $L(B_f)$ contains all ω-strings over Σ_{AP} satisfying f
- For any Kripke structure K, $L(B_K)$ contains all ω-strings over Σ_{AP} corresponding some computation path in K from an initial state
- It remains to check whether $L(B_K) \subseteq L(B_f)$
- Observe that $L(B_K) \subseteq L(B_f)$ if and only if $L(B_K) \cap \overline{L(B_f)} = \emptyset$

How to check $L(B_K) \cap \overline{L(B_f)} = \emptyset$?

- How to compute $\overline{L(B_f)}$?
- How to check $L(B_K) \cap \overline{L(B_f)} = \emptyset$?

Computing $\overline{L(B_f)}$

- Let M be a finite state automaton. Its *complement automaton*, $\overline{M}$, is a finite state automaton such that $L(\overline{M}) = \overline{L(M)}$
    - determize M and change the accepting states
- Can we do it for Büchi automata?
- Not directly. Deterministic Büchi automata is strictly less expressive than Büchi automata
    - A more general deterministic ω-automata is required
    - Alas, it is rather complicated

Computing $\overline{L(B_f)}$ (cont'd)

- Fortunately, there is an easy way out
- Observe that a computation path π satisfies f if and only if it does not satisfy $\neg f$
- Hence $\overline{L(B_f)} = L(B_{\neg f})$
  - Complementation of Büchi automata is not needed!

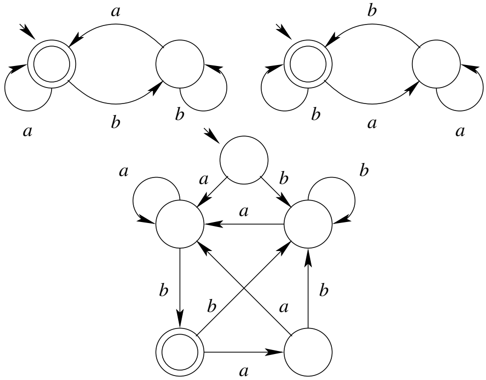
Checking $L(B_0) \cap L(B_1) = \emptyset$

- Let M^0 and M^1 be finite state automata
- How to check $L(M^0) \cap L(M^1) = \emptyset$?
  - construct product automaton $M^0 \times M^1$ and check if $L(M^0 \times M^1) = \emptyset$
- Can we do it for Büchi automata?
- Yes!

Product Automata $B^0 \times B^1$

- Let $B^0 = (\Sigma, Q^0, Q^0_0, \delta^0, F^0)$ and $B^1 = (\Sigma, Q^1, Q^1_0, \delta^1, F^1)$ be Büchi automata
- Define $B^0 \times B^1$ as follows.
  - Σ is its alphabets
  - $Q^0 \times Q^1 \times \{0, 1, 2\}$ are its states
  - $Q^0_0 \times Q^1_0 \times \{0\}$ are the initial states
  - $Q^0 \times Q^1 \times \{2\}$ are the accepting states
  - Moreover $\langle p^0, r^1, x \rangle \xrightarrow{a} \langle q^0, s^1, y \rangle$ if
    - $p^0 \xrightarrow{a}{}^0 q^0$ in B^0;
    - $r^1 \xrightarrow{a}{}^1 s^1$ in B^1; and
    - $y = \begin{cases} 1 & \text{if } x = 0 \text{ and } q^0 \in F^0 \\ 2 & \text{if } x = 1 \text{ and } s^1 \in F^1 \\ 0 & \text{if } x = 2 \\ x & \text{otherwise} \end{cases}$

Example of Product Automata



Product Büchi Automaton

Automata-Theoretic LTL Model Checking Algorithm

Input: a Kripke structure K and an LTL formula f

Output: whether $K \models f$

1. Construct B_K and $B_{\neg f}$ for K and $\neg f$ respectively
2. Check whether $L(B_K \times B_{\neg f}) = \emptyset$
   - if so, return *PASS*
   - otherwise, return *FAIL*

CTL Model Checking Problem

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure and f a CTL formula. We write $K \models f$ if $K, q_0 \models f$ for all $q_0 \in Q_0$
- Given a Kripke structure $K = (Q, Q_0, \delta, L)$ and a CTL formula f, the *CTL model checking problem* is to decide whether $K \models f$

Explicit-State CTL Model Checking

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure and f a CTL formula
- Let $Q' \subseteq Q$. Define

$$\begin{array}{rcl} Pre_K(Q') & = & \{q : \text{ there is a } q' \text{ such that } q \rightarrow q', q' \in Q'\} \\ PRE_K(Q') & = & \{q : \text{ for all } q' \text{ such that } q \rightarrow q', q' \in Q'\} \end{array}$$

- Define the function $[\![f]\!]_K$ as follows.

$$\begin{array}{rcl} [\![ap]\!]_K & = & \{q : ap \in L(q)\} \\ [\![\neg f]\!]_K & = & Q \setminus [\![f]\!]_K \\ [\![f \vee g]\!]_K & = & [\![f]\!]_K \cup [\![g]\!]_K \\ [\![\mathbf{EX} f]\!]_K & = & Pre_K([\![f]\!]_K) \\ [\![\mathbf{A}(f \mathbf{U} g)]\!]_K & = & [\![g]\!]_K \cup ([\![f]\!]_K \cap PRE_K([\![\mathbf{A}(f \mathbf{U} g)]\!]_K)) \\ [\![\mathbf{E}(f \mathbf{U} g)]\!]_K & = & [\![g]\!]_K \cup ([\![f]\!]_K \cap Pre_K([\![\mathbf{E}(f \mathbf{U} g)]\!]_K)) \end{array}$$

Solving $X = G(X)$

- A function $G : 2^Q \to 2^Q$ is *monotonic* if $A \subseteq B$ implies $G(A) \subseteq G(B)$
- Define $G_i \subseteq Q$ as follows

$$G_0 = \emptyset \text{ and } G_{i+1} = G(G_i)$$

- Facts
  - $G_i \subseteq G_{i+1}$ for all i
  - $G_i = G_{i+1}$ implies $G_j = G_i$ for all $j \geq i$
- Let $f \in \mathbb{N}$ be that $G_f = G_{f+1}$. Then $G_f = G_{f+1} = G(G_f)$. G_f is a *fixed point* of G
- Let $H \subseteq Q$ be that $H = G(H)$. Then $G_i \subseteq H$ for all i. Hence $G_f \subseteq H$. G_f is the *least fixed point* of G

Compute $[\![\mathbf{A}(f\mathbf{U}g)]\!]_K$

- Define
$$G(X) = [\![g]\!]_K \cup ([\![f]\!]_K \cap PRE_K(X))$$
- $G : 2^Q \to 2^Q$ is monotonic
- If Q is finite, there is an $f \in \mathbb{N}$ such that $G_f = G_{f+1}$
- Then $G_f = [\![\mathbf{A}(f\mathbf{U}g)]\!]_K$
- $[\![\mathbf{E}(f\mathbf{U}g)]\!]_K$ can be computed similarly

CTL Model Checking Algorithm

Input: a Kripke structure K and a CTL formula f

Output: whether $K \models f$

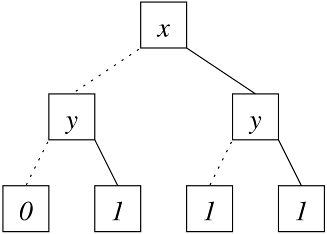
1. Compute $[\![f]\!]_K$
2. Check $Q_0 \subseteq [\![f]\!]_K$
   - if so, return *PASS*
   - otherwise, return *FAIL*

However, computing $[\![f]\!]_K$ is not easy when $|Q|$ is very large

Can we compute $[\![f]\!]_K$ efficiently (in practice)?

Decision Diagrams

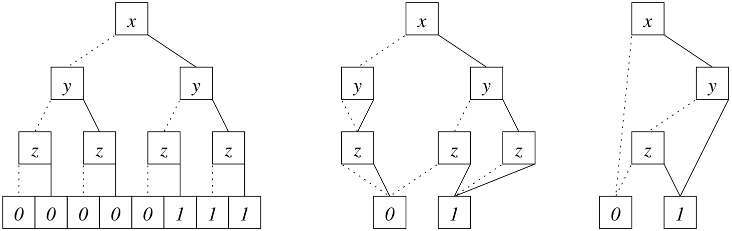
- Let $\mathbb{B} = \{\text{false}, \text{true}\}$ be the *Boolean domain*
- An *n-ary binary function* is a function from $\mathbb{B}^n$ to $\mathbb{B}$
- *Decision diagrams* are representations for binary functions



$f(x, y) = x \vee y$

From Decision Diagrams to Binary Decision Diagrams

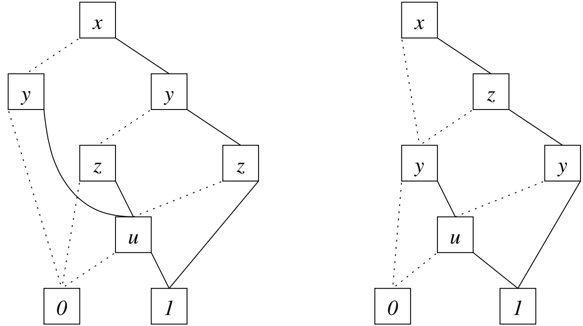
- Binary decision diagrams are obtained by
  - merging identical nodes
  - removing redundant nodes



$$f(x, y, z) = x \wedge (y \vee z)$$

Variable Order in BDD's

- For any fixed variable ordering, the BDD representation is canonical
  - $f = g$ if and only if $BDD(f) = BDD(g)$
- The size of BDD's depends on the order of variables
  - finding optimal order is NP-hard



$$(y \wedge u) \vee ((x \wedge z) \wedge (y \vee u))$$

BDD Operations

Let f and g be two n-ary binary functions. The following BDD operations are available:

- negation. **not** $BDD(f) = BDD(\neg f)$
- conjunction. $BDD(f)$ **and** $BDD(g) = BDD(f \wedge g)$
- disjunction. $BDD(f)$ **or** $BDD(g) = BDD(f \vee g)$
- universal quantification. **forall** $x_i BDD(f) = BDD(\forall x_i f)$
- existential quantification. **exists** $x_i BDD(f) = BDD(\exists x_i f)$
- renaiming. $BDD(f)[\overline{\mathbf{y}}/\overline{\mathbf{x}}] = BDD(f[\overline{x} := \overline{y}])$

Evaluating QBF by BDD's

- Given a Qualified Boolean Formula (QBF) f, one can evaluate f as follows
  - expand all qualifiers as mentioned
  - construct BDD for the expanded formula
  - return the resultant BDD (either false or true)
- BDD's can in fact determine the satisfiability or validity of any propositional logic formula

BDD's and CTL Model Checking

- Problems in the definition of $[\![f]\!]_K$ in explicit-state CTL model checking
  - Set operations
    - $[\![f \vee g]\!]_K = [\![f]\!]_K \cup [\![g]\!]_K$
  - $Pre_K(Q')$ and $PRE_K(Q')$
    - functions over sets of states
  - Fixed points
    - $[\![\mathbf{E}(f\mathbf{U}g)]\!]_K = [\![g]\!]_K \cup ([\![f]\!]_K \cap Pre_K([\![\mathbf{E}(f\mathbf{U}g)]\!]_K))$
- Can we solve them by BDD's?

BDD and Set Operations

- For simplicity, we only consider sets over binary vectors of size n
  - $\{000, 010, 100, 110\}$
- *Characteristic function* χ_H *for set* H is an n-ary binary function such that $\chi_H(x_n, x_{n-1}, \ldots, x_1) =$ true if and only if $x_n x_{n-1} \cdots x_1 \in H$
  - $\chi(x_3, x_2, x_1) = \neg x_1$
- Set operations correspond to logical operations
  - $\chi_{\overline{H_0}} = \neg \chi_{H_0}$
  - $\chi_{H_0 \cup H_1} = \chi_{H_0} \vee \chi_{H_1}$
  - $\chi_{H_0 \cap H_1} = \chi_{H_0} \wedge \chi_{H_1}$

Representing Transition Relations in BDD's

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure. For simplicity, assume $|Q| = 2^m$ for some m
- Each state $q \in Q$ can be represented by a binary vector of size m
- Each transition $q \rightarrow q'$ is represented by a pair of binary vector
- Hence the transition relation δ is represented by a set of binary vectors of size $2m$

Representing Transition Relations in BDD's (cont'd)

- Consider $K = (\{q_0, q_1, \ldots, q_7\}, \{q_0, q_2, q_4, q_6\}, \delta, L)$ with $q_i \longrightarrow q_{i+1 \text{ mod } 8}$
  - $q_0 = 000, q_1 = 001, \ldots, q_7 = 111$
  - $\chi_{Q_0}(x_3, x_2, x_1) = \neg x_1$
  - $\chi_\delta(x_3, x_2, x_1, x_3', x_2', x_1') =$

$$\left(\begin{array}{l} \neg x_3 \wedge \neg x_2 \wedge \neg x_1 \bigwedge \neg x_3' \wedge \neg x_2' \wedge x_1' \bigvee \\ \neg x_3 \wedge \neg x_2 \wedge x_1 \bigwedge \neg x_3' \wedge x_2' \wedge \neg x_1' \bigvee \\ \neg x_3 \wedge x_2 \wedge \neg x_1 \bigwedge \neg x_3' \wedge x_2' \wedge x_1' \bigvee \\ \neg x_3 \wedge x_2 \wedge x_1 \bigwedge x_3' \wedge \neg x_2' \wedge \neg x_1' \bigvee \\ x_3 \wedge \neg x_2 \wedge \neg x_1 \bigwedge x_3' \wedge \neg x_2' \wedge x_1' \bigvee \\ x_3 \wedge \neg x_2 \wedge x_1 \bigwedge x_3' \wedge x_2' \wedge \neg x_1' \bigvee \\ x_3 \wedge x_2 \wedge \neg x_1 \bigwedge x_3' \wedge x_2' \wedge x_1' \bigvee \\ x_3 \wedge x_2 \wedge x_1 \bigwedge \neg x_3' \wedge \neg x_2' \wedge \neg x_1' \end{array} \right)$$

- $\chi_\delta(b_3, b_2, b_1, b_3', b_2', b_1') = \text{true}$ if and only if $q_{(b_3 b_2 b_1)_2} \longrightarrow q_{(b_3' b_2' b_1')_2}$ where $(i)_2$ denotes the number represented by i in binary

Computing $Pre_K(Q')$ and $PRE_K(Q')$

- Q' is a set of binary vectors
- Recall
  $Pre_K(Q') = \{q : \text{ there is a } q' \text{ such that } q \to q', q' \in Q'\}$
- Let $\chi'_{Q'}$ be the characteristic function obtained by renaming each x to x' in $\chi_{Q'}$
  - Say, $\chi_{Q'}(x_3, x_2, x_1) = \neg x_1$. Then $\chi'_{Q'}(x'_3, x'_2, x'_1) = \neg x'_1$
- By assumption, $|Q| = 2^m$. Hence

$$
\begin{array}{rcl}
\chi_{Pre_K(Q')}(\overline{x}) & = & \exists \overline{x}'.\chi_\delta(\overline{x}, \overline{x}') \wedge \chi'_{Q'} \\
\chi_{PRE_K(Q')}(\overline{x}) & = & \exists \overline{x}'.\neg(\chi_\delta(\overline{x}, \overline{x}') \wedge \neg\chi'_{Q'})
\end{array}
$$

Solving $X = G(X)$ in BDD's

- X is a set of binary vectors and G is a set function over state sets
- χ_X can be represented by a BDD
- G can be computed by BDD operations
- We simply compute G_i iteratively

  Input: G a set function over state sets
  Output: G_f its least fixed point

  1. $i = 0$
  2. $G_i = \text{BDD}(\chi_\emptyset)$
  3. do
  4. $\quad i = i + 1$
  5. $\quad G_i = G(G_{i-1})$
  6. while $G_i \neq G_{i-1}$
  7. return G_i

Symbolic CTL Model Checking

- Given $K = (Q, Q_0, \delta, L)$
- Encode δ and L in BDD's
  - $\chi_a(\overline{x}) = 1$ if and only if $a \in L(\overline{x})$
- Compute BDD($\chi_{[\![f]\!]_K}$)
- Check if BDD($\chi_{Q_0} \wedge \neg\chi_{[\![f]\!]_K}$) $=$ BDD($\chi_{\emptyset}$)
  - if so, return *PASS*
  - otherwise, return *FAIL*

Limitation of BDD's

- Symbolic CTL model checking does not solve all our problems
  - BDD's are hard to predicate
    - the size is very sensitive to variable ordering
  - BDD's cannot handle real systems
    - up to 300 binary variables
  - Oftentimes, BDD's would blow up while building transition relations
    - no information at all when it doesn't work
- Techniques that can be scaled up are always needed

ACTL

- A CTL formula is in *negative normal form* if the negation appears only before atomic propositions
  - For instance, $\mathbf{AF}\neg p$ is in nnf but $\neg\mathbf{EG}p$ is not
- Write $K, \pi \models f\mathbf{R}g$ if for all $j \geq 0$, for all $i < j$ $K, \pi(i) \not\models f$ implies $K, \pi(j) \models g$
  - Observe that $f\mathbf{R}g \equiv \neg(\neg f\mathbf{U}\neg g)$
- All CTL formula can be transformed to its negative normal form
  - Use $\neg\neg f \equiv f$, $\neg(f \vee g) \equiv \neg f \wedge \neg g$, $\neg(f \wedge g) \equiv \neg f \vee \neg g$, $\neg\mathbf{AX}f \equiv \mathbf{EX}\neg f$, $\neg\mathbf{EX}f \equiv \mathbf{AX}\neg f$, $\neg\mathbf{E}(f\mathbf{U}g) \equiv \mathbf{A}(\neg f\mathbf{R}\neg g)$, $\neg\mathbf{A}(f\mathbf{U}g) \equiv \mathbf{E}(\neg f\mathbf{R}\neg g)$
- *ACTL* is a subclass of CTL, where only universal path quantifier is allowed in negative normal form
  - $\mathbf{AG}p$ and $\neg\mathbf{E}(f\mathbf{U}g)$ are in ACTL but $\mathbf{EG}p$ and $\mathbf{AGEF}p$ are not

Satisfiability and Validity

- Consider a propositional logic formula, say, $[p \rightarrow (q \vee r)] \wedge [q \vee \neg r]$
- A *truth assignment* is a mapping from propositional variables (p, q, r, etc) to Boolean domain
- A propositional logic formula is *satisfiable* if there is a truth assignment that makes the formula evaluate to true
  - For instance, the above formula can be satisfied by setting $p =$ false, $q =$ true, and $r =$ true
- A propositional logic formula is *valid* if for all truth assignment, the formula evaluates to true
  - For instance, the above formula evaluates to false when $p =$ true, $q =$ false, and $r =$ false. It is not valid
- For any propositional formula f, f is not satisfiable if and only if $\neg f$ is valid

Boolean Satisfiability

- Given a propositional logic formula, determine whether there is a satisfying truth assignment
- First NP-complete problem
- Since mid 90's, many practical SAT solvers are available
  - by "practical", we mean SAT solvers that can handle thousands of binary variables!
  - widely used SAT solvers are MiniSAT, zchaff, grasp
- We will use SAT solvers to solve ACTL model checking within bounded steps

Bounded Model Checking

- The idea is to verify the Kripke structure up to a fixed number of steps
- Equivalently, bounded model checking aims to find bugs within a fixed number of steps
  - if bugs are found, bounded model checker reports them
  - if bugs cannot be found in the first n steps, it does not guarantee the correctness of the Kripke structure

SAT and Bounded Model Checking

- Consider the formula $\mathbf{AX}p$ on the Kripke structure K
- What is a bug in K?
- By definition $K \not\models \mathbf{AX}p$ if $K, q_0 \models \neg\mathbf{AX}p$ for some $q_0 \in Q_0$
- Hence our goal is to find a $q_0 \in Q_0$ such that $K, q_0 \models \mathbf{EX}\neg p$
- Can it be done by SAT solvers?
- Yes! Checking the satisfiability of the following formula suffices.
  - $\chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \neg\chi_p(\overline{x}_1)$
- What about verifying $\mathbf{EX}p$?
  - Not directly. Checking the satisfiability of $[\chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1)] \rightarrow \neg\chi_p(\overline{x}_1)$ does not work. Why?

SAT Solvers and ACTL Bounded Model Checking

- Let f be an ACTL formula
- $\neg f$ is equivalent to a CTL formula where only existential path quantifiers occur
  - for instance, $\neg \mathbf{AGAF} p \equiv \mathbf{EFEG} \neg p$
- It suffices to find a $q_0 \in Q_0$ such that $K, q_0 \models \neg f$
  - if so, a bug is found and can be reported
  - if not, we conclude there is no bug up to the bound

Bounded ACTL Model Checking – Example

- Let $K = (Q, Q_0, \delta, L)$ be a Kripke structure with $|Q| = 2^m$
- Consider verifying $K \models \mathbf{AG}a$ up to the first 3 steps
- We hence try to find a $q_0 \in Q_0$ such that $K, q_0 \models \mathbf{EF}\neg a$
- Consider the following propositional formula

$$
\begin{aligned}
& F_3(\overline{x}_0, \overline{x}_1, \overline{x}_2, \overline{x}_3) \\
& \quad = \quad \chi_{Q_0}(\overline{x}_0) \wedge \neg\chi_a(\overline{x}_0) \bigvee \\
& \qquad\qquad \chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \neg\chi_a(\overline{x}_1) \bigvee \\
& \qquad\qquad \chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \chi_\delta(\overline{x}_1, \overline{x}_2) \wedge \neg\chi_a(\overline{x}_2) \bigvee \\
& \qquad\qquad \chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \chi_\delta(\overline{x}_1, \overline{x}_2) \wedge \chi_\delta(\overline{x}_2, \overline{x}_3) \wedge \neg\chi_a(\overline{x}_3)
\end{aligned}
$$

- Then $F_3(\overline{x}_0, \overline{x}_1, \overline{x}_2, \overline{x}_3)$ is satisfiable if and only if there is a state q reachable from some $q_0 \in Q_0$ in three steps such that $a \notin L(q)$.

Notes about ACTL Bounded Model Checking

- Pros
  - Partial information. Even though we cannot verify the system, we do know it is correct up to a certain number of steps
  - Scalability. Modern SAT solvers can handle thousands of binary variables. We can check larger systems
- Cons
  - A bit tricky to verify systems for sure. Extending bounded model checking to model checking is not straightforward
  - Does not work well for general CTL formulae. Alternation of universal and existential path quantifiers causes problems

From Bounded to Unbounded Model Checking

- It is a bit tricky to verify ACTL by SAT solvers completely
- We will introduce a complete SAT-based verification algorithm for invariant checking
- An *invariant* is an atomic proposition which is satisfied in all states reachable from initial states
    - a is an invariant if and only if $\mathbf{AG}a$ and $\mathbf{G}a$ hold
- We will apply inductive reasoning in invariant checking!

Induction

- Consider verifying $\mathbf{AG}a$ on $K = (Q, Q_0, \delta, L)$
- Suppose we know the following
  - $a \in L(q_0)$ for all $q_0 \in Q_0$
  - for all q and q' such that $q \longrightarrow q'$, $a \in L(q)$ implies $a \in L(q')$
- Can we conclude $K \models \mathbf{AG}a$?
  - Yes!

Proof.

If $K \not\models \mathbf{AG}a$, there is a $q_0, q_1, \ldots, q_m \in Q$ such that

- $q_0 \in Q_0$
- $q_i \longrightarrow q_{i+1}$ for $0 \leq i < m$
- $a \in L(q_i)$ for $0 \leq i < m$ but $a \notin L(q_m)$

Then $q_m \notin Q_0$ by the basis. Moreover, $a \in L(q_m)$ for $a \in L(q_{m-1})$ and $q_{m-1} \longrightarrow q_m$ by inductive step $\square$

From Induction to k-Induction

- The idea can be generalized to more than one step
  - $a \in L(q_i)$ for all $q_i \in Q_i$ and $0 \leq i < k$ where

    $$Q_i = \{q' : q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_i \text{ for some } q_0 \in Q_0\}$$

  - $a \in L(q_i)$ for $0 \leq i < k$ implies $a \in L(q_k)$ where $q_i \longrightarrow q_{i+1}$ for $0 \leq i < k$
- How can we perform k-induction by SAT solvers

Induction by SAT Solvers

- Consider the following two SAT problems
  - $\chi_{Q_0}(\overline{x}_0) \wedge \neg\chi_a(\overline{x}_0)$
  - $\chi_a(\overline{y}_0) \wedge \chi_\delta(\overline{y}_0, \overline{y}_1) \wedge \neg\chi_a(\overline{y}_1)$
- What do they mean if they are not satisfiable?
  - it's impossible to have an initial state not satisfying a
    - all initial states satisfy a
  - it's impossible to reach a state not satisfying a from a state satisfying a
    - any state satisfying a can only go to states satisfying a
- Hence, if these propositional logic formulae are unsatisfiable, we conclude a is an invariant

k-Induction by SAT Solvers

- The technique can be generalized to k-induction
- Consider the following propositional logic formulae
  - $\chi_{Q_0}(\overline{x}_0) \wedge \neg\chi_a(\overline{x}_0)$
  - $\chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \neg\chi_a(\overline{x}_0)$
  - $\cdots$
  - $\chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \cdots \chi_\delta(\overline{x}_{k-2}, \overline{x}_{k-1}) \wedge \neg\chi_a(\overline{x}_{k-1})$
  - $\chi_a(\overline{y}_0) \wedge \chi_\delta(\overline{y}_0, \overline{y}_1) \bigwedge \chi_a(\overline{y}_1) \wedge \chi_\delta(\overline{y}_1, \overline{y}_2) \bigwedge \cdots \chi_a(\overline{y}_{k-1}) \wedge \chi_\delta(\overline{y}_{k-1}, \overline{y}_k) \bigwedge \neg\chi_a(\overline{y}_k)$
- If all of them are unsatisfiable, we conclude a is an invariant
  - what if some of them are satisfiable?

From k-Induction to $k+1$-Induction

- When k-induction fails, there are two possibilities
  - some of basis formulae are satisfiable
    - $\chi_{Q_0}(\overline{x}_0) \wedge \neg\chi_a(\overline{x}_0)$, $\chi_{Q_0}(\overline{x}_0) \wedge \chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \neg\chi_a(\overline{x}_0)$, ...
    - a counterexample is found!
  - the inductive formula is satisfiable
    - $\chi_a(\overline{y}_0) \wedge \chi_\delta(\overline{y}_0, \overline{y}_1) \bigwedge \chi_a(\overline{y}_1) \wedge \chi_\delta(\overline{y}_1, \overline{y}_2) \bigwedge \cdots \chi_a(\overline{y}_{k-1}) \wedge \chi_\delta(\overline{y}_{k-1}, \overline{y}_k) \bigwedge \neg\chi_a(\overline{y}_k)$
- If the inductive step is satisfiable, one increases k and performs $k+1$-induction
  - if a is not an invariant, there is a k such that k-induction fails in the basis
  - the basis will be satisfiable for some k
  - what if a is indeed an invariant?
    - can we always establish invariance by induction? not necessarily!

From Induction to Complete Induction

- If the basis formulae are not satisfiable but the inductive formula is satisfiable, when can we conclude the invariant checking passes?
- Idea: the shortest counterexample cannot be longer than the diameter of reachability graph
  - The reachability graph consists of states as nodes and transitions as edges

Proof.

Let k be the diameter of reachability graph. Consider $q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_k \longrightarrow q_{k+1}$. Then $q_i = q_j$ for some $0 \leq i < j \leq k+1$. Hence $q_0 \longrightarrow q_1 \longrightarrow \cdots \longrightarrow q_i \longrightarrow q_{j+1} \longrightarrow \cdots \longrightarrow q_{k+1}$ is a shorter computation path to q_{k+1} $\square$

From Induction to Complete Induction (cont'd)

- It suffices to find the diameter of reachability graph
- Consider the following formula

$$
\begin{array}{l}
\chi_{Q_0}(\overline{x}_0) \bigwedge \\
\chi_\delta(\overline{x}_0, \overline{x}_1) \wedge \overline{x}_1 \neq \overline{x}_0 \bigwedge \\
\chi_\delta(\overline{x}_1, \overline{x}_2) \wedge \overline{x}_2 \neq \overline{x}_0 \wedge \overline{x}_2 \neq \overline{x}_1 \bigwedge \\
\ldots \\
\chi_\delta(\overline{x}_{k-1}, \overline{x}_k) \wedge \overline{x}_k \neq \overline{x}_0 \wedge \overline{x}_k \neq \overline{x}_1 \wedge \cdots \wedge \overline{x}_k \neq \overline{x}_{k-1}
\end{array}
$$

- If the formula is unsatisfiable for some k, we know the diameter of reachability graph is $k-1$

Complete SAT-based Invariant Checking

- Here is the algorithm

  Input: $K = (Q, Q_0, \delta, L)$ and an atomic proposition a
  Output: whether a is an invariant in K

  1. $k := 1$
  2. loop
  3.   perform k-induction
  4.   if a counterexample is found, return *FAIL*
  5.   if the diameter is k, return *PASS*
  6. $k := k + 1$

Wrap-up

We have introduced

- both LTL and CTL
- an automata-theoretic LTL model checking algorithm
- a BDD-based CTL model checking algorithm
- a SAT-based invariant checking algorithm
- SPIN and NUSMV

Current Research

- Finite-state models to infinite-state models
  - context-free processes and pushdown systems
- Proof theory + model checking = ?
- Computational learning theory
- SAT-based model checking algorithm for universal μ-calculus