State Machines
Formal Methods
Lecture 3

Farn Wang
Dept. of Electrical Engineering
National Taiwan University

Purpose

Understanding
- the formal semantics of programs
- the definition of state transition systems
- Problems of finite-state system analysis
- Algorithms for the problems
Organization

- Sequential Program – Operational Semantics
- Kripke Structures
- Concurrent Systems
- Verification problems of state-transition systems

State-space representations from programs - States, transitions

- Program Variables
  - Program Counter (pc), data variables, ...
- Program State
  - Valuation of program variables
- Transition
  - Moving one state to another by executing a program statement.
Kripke structure from programs - operational Semantics

- Operational Semantics clarifies the execution of a program.
- Closes the gap between the text of a program and the behaviors represented by it.
- Let us look only at sequential programs for the moment.

IMP : a toy imperative language

- IMP is an imperative language in the style of PASCAL or C (even though some of the syntax may be different)
- The language contains arithmetic and boolean expressions as well as if-then-else, while statements.
- The syntax of the program will be described by BNF grammars.
IMP : a toy imperative language

- During execution of IMP program, the state of execution will be captured by the values of program variables.
- Operational semantics will be described by rules which specify how
  - Expressions in IMP pgm. are evaluated
  - Statements in IMP pgm. change the state

BNF, syntax definitions

Note!
Be sure how to read BNF!
- used for define syntax of context-free language
- important for the definition of
  - automata predicates and
  - temporal logics
- Used throughout the lectures!
- In exam: violate the syntax rules ➔ no credit.

A ::= c | x | (M) | A₁ + A₂ | A₁ - A₂
M ::= c | x | (A) | M₁ * M₂ | M₁ / M₂
  c is an integer
  x is a variable name.
BNF, syntax definitions
- Examples of context-sentivity

Session 1:
- A: Are you married?
- B: No!
- A: Do you have children?
- B: 😞

Session 3:
- A: Are you married?
- B: Yes!
- A: Do you have children?
- B: 😊

Rude contextual interpretation: Are you a single parent?

Session 2:
- A: Do you have children?
- B: Yes!
- A: Are you married?
- B: 😞

Session 4:
- A: Do you have children?
- B: No!
- A: Are you married?
- B: 😊

Rude contextual interpretation: Are you a single parent?

BNF, syntax definitions

A ::= c | x | (M) | A_1 + A_2 | A_1 - A_2
M ::= c | x | (A) | M_1 * M_2 | M_1 / M_2

c is an integer
x is a variable name.

(3*x) + y - 3
BNF, syntax definitions
- derivation trees (from top down)

A ::= c | x | (M) | A_1 + A_2 | A_1 - A_2
M ::= c | x | (A) | M_1 * M_2 | M_1 / M_2

- A is an integer
- x is a variable name.

used in string generation.

(3*x) + y - 3

BNF, syntax definitions
- parsing trees (from bottom up)

A ::= c | x | (M) | A_1 + A_2 | A_1 - A_2
M ::= c | x | (A) | M_1 * M_2 | M_1 / M_2

- A is an integer
- x is a variable name.

used in compiler.

(3*x) + y - 3
BNF, another syntax definition

\[
A ::= c \mid x \mid LM R \mid A_1 \cdot P \cdot A_2 \\
M ::= c \mid x \mid LA R \mid M_1 \cdot K \cdot M_2 \\
L ::= '(' \\
R ::= ')' \\
P ::= '+' \mid '-' \\
K ::= '*' \mid '/'
\]

c is an integer
x is a variable name.

Syntax of IMP

- Non-negative integers \( N \)
- Truth values \( T = \{\text{true}, \text{false}\} \)
- Variables \( V \)
- Arithmetic expressions \( A \)
- Boolean expressions \( B \)
- Statements/commands \( C \)
Syntax of expression

**Arithmetic expressions**

\[
A ::= c \mid x \mid A_1 \oplus A_2 \mid (A) \mid (B)?A_1:A_2
\]

\(c \in \mathbb{R}\), \(x\) is a variable.

\(\oplus \in \{+, -, \ast, \div, \%\}\)

**Boolean expressions**

\[
B ::= \text{true} \mid A_1 \approx A_2 \mid \neg B_1 \mid B_1 \lor B_2 \mid (B_1)
\]

\(\approx \in \{\leq, <, ==, !=, >, \geq\}\)

\(false \equiv \neg true, B_1=>B_2 \equiv (\neg B_1)||B_2,\)

\(B_1&&B_2 = \neg((\neg B_1)||\neg B_2)\)

Expressions
- examples

- \(x + 2y < 3\)
- \(x*y + y*y*3 == z\)
- \(x\)
- \(x + 2y < 3 \mid \mid x*y + y*y*3 <= z\)
- \((x + 2*y < 3 \mid \mid \sim*x*y + y*y*3 == z)\)&&\text{flag}\)

Please construct the parsing trees.
Syntax of Commands $C$

\[
C ::= ; \\
| x = A; \\
| \{ C_1 \} \\
| C_1 C_2 \\
| \text{if (B) } C_1 \text{ else } C_2 \\
| \text{while (B) } C_1
\]

---

**IMP**

- statement example

```plaintext
w = 0;
x = 0;
y = z*z;
while (x < y) {
    w = w + x*z;
x = x + 1;
}
if (w > z*z*z) w = z*z*z;
```

Please construct the parsing tree.
Execution model

- Operational semantics of IMP describes how programs in that language are executed.
- To describe this, it needs to assume an underlying execution model.
- The execution model could be thought as a state machine although not necessarily a finite state machine.

Operational Semantics

Operational Semantics for the IMP language will give rules to describe the following:
- Give a state \( s \)
- How to evaluate arithmetic expressions
- How to evaluate Boolean expressions
- How the commands can alter \( s \) to a new state \( s' \)
States

A state is a valuation of program variables i.e. each variable is mapped to a value in its type.

Thus, if \{a,b\} are the only variables in an IMP program, then each of the following are states in the execution model:
- a=0, b=0
- a=0, b=1
- a=0, b=2
- ...
- a=1, b=0
- ...

Meaning of Arith. Expressions (1)

\(\langle A, s \rangle\)

- Numbers: \(\langle c, s \rangle = c\)
  Number \(c\) in any state \(s\) evaluates to \(c\)
  
  *E.g. \(\langle 0, s \rangle = 0, \langle 5, s \rangle = 5\)*

- Variables: \(\langle x, s \rangle = s(x)\)
  \(\langle X, s \rangle = s(X)\)
  Variable \(X\) in state \(s\) evaluates to value of \(x\) in \(s\).
  
  *E.g. \(\langle a, (a = 5, b = 20) \rangle = 5, \langle b, (a = 5, b = 20) \rangle = 20\)*
Meaning of Arith. Expressions (2)

\[ \langle A, s \rangle \]

- **Sums:** \( \langle a + b, s \rangle = \langle a, s \rangle + \langle b, s \rangle \)
  
  e.g. \( \langle a + b, (a =5, b=20) \rangle = 25 \)

- **Products:** \( \langle a \ast b, s \rangle = \langle a, s \rangle \ast \langle b, s \rangle \)
  
  e.g. \( \langle a \ast b, (a =5, b=20) \rangle = 100 \)

Example arith. expr. evaluation

Evaluating meaning of a complicated arithmetic expression will require

- Several application of the above rules
- Operator precedence

EX: \( \langle a \ast b + b, (a =5, b=20) \rangle \)

\[
= \langle a \ast b, (a =5, b=20) \rangle + \langle b, (a =5, b=20) \rangle \\
= \langle a, (a =5, b=20) \rangle \ast \langle b, (a =5, b=20) \rangle + 20 \\
= 5 \ast 20 + 20 = 120
\]
Meaning of Boolean Expression (1)

\( \langle B, s \rangle \)
- \( \langle \text{true}, s \rangle = \text{true} \)
- \( \langle \text{false}, s \rangle = \text{false} \)
- Inequality Check:
  \( \langle A_1 \approx A_2, s \rangle = \langle A_1, s \rangle \approx \langle A_2, s \rangle \)
- Negation:
  \( \langle \sim B, s \rangle = \sim \langle B, s \rangle \)
- Disjunction:
  \( \langle B_1 \| B_2, s \rangle = \langle B_1, s \rangle \| \langle B_2, s \rangle \)

Workout

State s: \((a=5, b=6)\)
- \( \langle a=b, s \rangle \equiv \langle a, s \rangle = \langle b, s \rangle \equiv 5=6 \equiv \text{false} \)
- \( \langle \sim a=b, s \rangle = \sim \langle a=b, s \rangle = \text{true} \)
- \( \langle a<=b, s \rangle \equiv \langle a, s \rangle <= \langle b, s \rangle \equiv 5<=6 \equiv \text{true} \)
- \( \langle a<=b \&\& a=b, s \rangle \)
  \( = \langle a<=b, s \rangle \&\& \langle a=b, s \rangle \)
  \( = \text{true} \&\& \text{false} \)
  \( = \text{false} \)
Meaning of Expressions

- Expressions evaluate to values in a given state
- Therefore, the meaning of expressions are given by values.
  - Boolean values for boolean expressions
  - Number for arithmetic expressions
- Using the meaning of expressions, we can assign meaning to commands.

Workout

State s: (a=3, b=10, c=5)

1. \langle a+3*b*c, s \rangle = t
2. \langle a+3*b=c, s \rangle = f
3. \langle \neg, s \rangle = f
4. \langle \neq, s \rangle = t
5. \langle \wedge, s \rangle = t
6. \langle \vee, s \rangle = f
7. \langle \leq, s \rangle = f
8. \langle \gt, s \rangle = f
9. \langle \land, s \rangle = t
10. \langle \neq, \land, s \rangle = t
11. \langle \rightarrow, s \rangle = f
12. \langle \lor, s \rangle = t
13. \langle \rightarrow, s \rangle = t
Meaning of Commands

- Execution of commands leads to a change of program state.
- Therefore the meaning of a command $C$ is: If $C$ is executed in some state $s$, how does it change $s$ to $s'$.

$$\langle C, s \rangle = s'$$

Rules for commands (1)

- $\langle ;, s \rangle = s$
- $\langle x=A; s \rangle = s[x=A]$
  - $s[x=A]$ is the same as state $s$ except that the value of $x$ is $\langle A, s \rangle$.
  - Ex: $(a=5,b=20,c=2)[a=7] = (a=7,b=20,c=2)$
  - Ex: $(a=5,b=20,c=2)[a=5] = (a=5,b=20,c=2)$
  - Ex: $(a=5,b=20,c=2)[a=b+c] = (a=22,b=20,c=2)$
- $\langle \{ C_1 \}, s \rangle = \langle C_1, s \rangle$
Rules for commands (1)

\[ \langle C, s \rangle \]
- \[ \langle C_1C_2, s \rangle = \langle C_2, \langle C_1, s \rangle \rangle \]

- \[ \langle \text{if} \ (B) \ C_1 \ 	ext{else} \ C_2 \ , \ s \rangle = \langle C_1, s \rangle \] if \[ \langle B, s \rangle = true \]
  \[ \langle \text{if} \ (B) \ C_1 \ 	ext{else} \ C_2 \ , \ s \rangle = \langle C_2, s \rangle \] if \[ \langle B, s \rangle = false \]

- \[ \langle \text{while}(B)C_1, s \rangle = s \] if \[ \langle B,s \rangle = false \]
  \[ \langle \text{while}(B)C_1, s \rangle = \langle \text{while}(B)C_1, \langle C_1,s \rangle \rangle \] if \[ \langle B,s \rangle = true \]

Summary of rules

- The meaning of each commands specifies how an execution of the command changes state.
- Roughly speaking, this is done by simulating the execution of the commands.
- For example, the rule for while essentially unfolds the iterations of while loop.
Kripke Structure

- A state-transition system that captures
  - What is true of a state
  - What can be viewed as an atomic move
  - The succession of states

- Static representation that can be unrolled to a tree of execution traces, on which temporal properties are verified

---

Saul Kripke

Born in 1940

Professor, Princeton University

Distinguished Professor, CUNY

- Kripke semantics
- Naming & Necessity in the philosophy of languages
- Interpretation of the philosophy of Wittgenstein
- Theory of truth

- wrote his first essay on Kripke structure at 16
- invited to teach at Princeton
- taught a graduate logic course at MIT since sophomore year at Harvard.

I'm honored by your proposal, by my mum says I have to finish high-school first.
Kripke structure - syntax
A = (S, S₀, R, L)
- S
  - a set of all states of the system
- S₀ ⊆ S
  - a set of initial states
- R ⊆ S × S
  - a transition relation between states
- L : S → 2^P
  - a function that associates each state with set of propositions true in that state

• L allows us to describe the truth/falsehood of a proposition in the various states of a system.
• The propositions refer to valuations of the state variables.

Kripke Model - syntax
- Set of states S={q₁,q₂,q₃}
- Set of initial states S₀={q₁}
- R ={(q₁,q₂),(q₂,q₂),
  (q₁,q₃),(q₃, q₁),
  (q₃,q₂)}

- Set of atomic propositions AP={a,b}
- L(q₁)={a}, L(q₂)={a,b}, L(q₃)={b}
Kripke structure
- semantics

Given a Kripke structure $A = (S, S_0, R, L)$, a run is a finite or infinite sequence

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

such that

- $s_0 \in S_0$
- for each $k \in \mathbb{N}$, if $s_{k+1}$ exists,  
  - $s_{k+1} \in S$ and  
  - $R(s_k, s_{k+1})$ is true.

Control and data variables

- State = valuation of control and data vars.
- In our example  
  - $pc0$, $pc1$ are control variables.  
  - $turn$ is a shared data variable.  
- To generate a finite state transition system  
  - Data variables must have finite types, and  
  - Finitely many control locations
Program $\rightarrow$ Kripke structure
- Data variables
Data variables often do not have finite types
- integer, …
- Usually \textit{abstracted} into a finite type.
- An integer variable can be abstracted to \{-,0,+\}
- Just store the information about the sign of the variable. (coming up with these abstractions is a whole new problem).

Program $\rightarrow$ Kripke structure
- Control Locations
Isn’t the control locations of a program always finite ?
- NO, because your program may be a concurrent program with unboundedly many processes or threads (\textit{parameterized system}).
- Can employ control abstractions (such as symmetry reduction)
Program $\rightarrow$ Kripke structure  
- States and Transitions

- Each component makes a move at every step.
- Digital circuits are most often *synchronous*.
  - Common clock driving the system.
  - Contents of flip-flops define the states.
  - On every clock pulse, the content of every flip-flop (potentially) changes.
- This change is captured by the transition relation.
Program → Kripke structure
- States and Transitions

- Define $V = \{v_1, \ldots, v_n\}$, boolean variables representing state of flip-flops in the circuit.
- Set of states represented by boolean formula over $v_1, \ldots, v_n$.
- To define transitions, define a fresh set of variables $V' = \{v'_1, \ldots, v'_n\}$. These are the next state variables.
- The transitions are now represented by a relation $R(V, V') \subseteq V \times V'$

Kripke structure
- Transition Relation

- $(s, s') \in R(V, V')$ implies $s \rightarrow s'$
- Now, $R(V, V') = \bigcup_{i \in \{1, \ldots, n\}} R_i(V, V')$, where $R_i$ captures the changes in state variable $v_i$
- Define $R_i(V, V') = (v'_i \Leftrightarrow f_i(V))$ where $f_i(V)$ is a boolean function defining the value of flip-flop $i$ in next state.
- Given a synchronous circuit, we then need to define $f_i(V)$ for each $i$. 

Transition relation
- A synchronous mod 8 counter
- \( V = \{v_2, v_1, v_0\} \), where \( v_0 \) is the least significant bit.
- The transitions can be enumerated as:
  \( 000 \rightarrow 001 \rightarrow 010 \rightarrow \ldots \)
- Alternatively define how each of the three bits are changed on every clock cycle
  \( v_0' = \neg v_0 \) (the least significant bit)
  \( v_1' = v_0 \oplus v_1 \)
  \( v_2' = (v_0 \land v_1) \oplus v_2 \) (the most significant bit)

Kripke Structure
- example
Suppose there is a program

```
initially x==1 && y==1;
while (true)
  x = (x+y) % 2;
```

where \( x \) and \( y \) range over \( D = \{0,1\} \)
Kripke Structure
- example

Suppose there is a program

<table>
<thead>
<tr>
<th>Initially</th>
<th>While (true)</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x==1 &amp;&amp; y==1;</td>
<td>x = (x+y) % 2;</td>
<td></td>
</tr>
</tbody>
</table>

where x and y range over \( D=\{0,1\} \)

\[ S=DxD=\{(0,0),(0,1),(1,0),(1,1)\} \]
\[ S_0=\{(1,1)\} \]
\[ R=\{((1,1),(0,1)),((0,1),(1,1)),((1,0),(1,0)),((0,0),(0,0))\} \]
\[ L((1,1))=\{x=1,y=1\} \]
\[ L((0,1))=\{x=0,y=1\} \]
\[ L((1,0))=\{x=1,y=0\} \]
\[ L((0,0))=\{x=0,y=0\} \]

where x and y range over \( D=\{0,1\} \)
Suppose there is a program

\[
\text{initially } x = 1 \land y = 1; \\
\text{while (true)} \\
x = (x + y) \mod 2;
\]

where \( x \) and \( y \) range over \( D = \{0,1\} \)

---

**Workout**

**- Kripke Structure**

Suppose there is a program

\[
\text{initially } x = 1 \land y = 1; \\
\text{while (true)} \\
x = (x + y) \mod 3;
\]

where \( x \) and \( y \) range over \( D = [0,2] \)
Kripke Structure
- an example

Initially x=0
While (true)
x:=1-x;

A 2-bit counter operates at bit-level.
Kripke Structure - workout

Write a simple program for the Kripke structures in the last page.

Automata & Kripke structure
State-transition graphs
- an extension of automata for complex models

State-transition graphs

True guards are omitted
Null actions are omitted
State transition graphs
- from a program

(1) f(n) {
(2)   x = 1;
(3)   while (n > 0) {
(4)       x = x * 2; n = n - 1;
(5)   }
(6)   return x;
(7) }

State-transition graphs
- from a procedure call example

(1) f(n) {
(2)   x = 1;
(3)   while n > 0, do {
(4)       x = x * 2; n = n - 1;
(5)   }
(6)   return x;
(7) }
(8)   main (c,d) {
(9)       c = f(c + d);
(10)      if (c > 10)
(11)         print "yes".
(12)      else print "no".
(13)   }
Guarded commands with modes (GCM)

- A text language for state-transition graphs
- For multi-thread systems
- Extension with programming concepts

<table>
<thead>
<tr>
<th>Process count declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
</tr>
<tr>
<td>Inline expression declaration (optional)</td>
</tr>
<tr>
<td>Mode declaration</td>
</tr>
<tr>
<td>Initial condition</td>
</tr>
<tr>
<td>Specification (optional)</td>
</tr>
</tbody>
</table>

V is a variable declaration.
E is an arithmetic expression.
B is a Boolean condition.
C is a program of IMP commands or “goto name” where name is a mode name.

Each rule R is executed atomically.
- for the modeling of complex behaviors in transitions.
A program can be a set of GCM.
- At any moment, at most one command is executed.
Guarded commands with modes (GCM) - a language for state-transition graphs

\[
G ::= P \ VS \ [ILS] \ MS \ INI \ [SP]
\]

\[
P ::= \text{process count} = E;
\]

\[
VS ::= | V VS
\]

\[
V ::= \text{SCOPE TYPE } x [::c..c]; // c \in N, x \ a \ variable
\]

\[
SCOPE ::= \text{global} | \text{local}
\]

\[
TYPE ::= \text{discrete} | \text{pointer} | \text{clock} | \text{dense}
\]

\[
| \text{synchronizer}
\]

Threads are indexed 1 through c.

---

Guarded commands with modes (GCM) - a language for state-transition graphs

\[
IL ::= \text{inline TYPE name (FSL) } \{ EI \}
\]

\[
FSL ::= | FS
\]

\[
FS ::= f | f, FS // f: a formal argument
\]

\[
EI ::= f | x | x[c] // x: a declared discrete variable
\]

\[
| ( EI ) | EI+EI | EI-EI | EI*EI | EI/EE | EI%EI
\]

\[
| ( BI ) ? EI : EI | #PS | P
\]

\[
| name ( EISS )
\]

\[
EISS ::= | EIS
\]

\[
EIS ::= EI | EI, EIS
\]
Guarded commands with modes (GCM)
- a language for state-transition graphs

\[ BI ::= ( BI ) | EI < E | EI < E | EI > E | EI > E \]
\[ | EI = E | EI /= E \]
\[ | BI && BI | BI \| BI | ~BI | BI => BI \]
\[ | \forall x : c .. c, BI \] exists x : c .. c, BI
\[ | name ( EISS ) \]

Guarded commands with modes (GCM)
- a language for state-transition graphs

\[ MS ::= | M MS \]
\[ M ::= [ urgent ] mode name (B) \{ RS \} \]
\[ B ::= (B) | E < E | E < E | E > E | E > E | E = E | E /= E \]
\[ | B && B | B \| B | ~B | B => B E | name ( ESS ) \]
\[ | \forall x : c .. c, B \] exists x : c .. c, B
\[ E ::= x | x[c] \] // x : a declared discrete variable
\[ | (E) | E + E | E - E | E * E | E / E | E \% E \]
\[ | (B)?E : E | name ( ESS ) \]
\[ ESS ::= | ES \]
\[ ES ::= E | E , ES \]
Guarded commands with modes (GCM) - a language for state-transition graphs

\[
\begin{align*}
RS ::= & \mid R RS \\
R ::= & \text{when } SS \ (B) \ \text{may } C \\
SS ::= & \mid S SS \\
S ::= & \ ?x \mid ?(E)x \mid bx \mid !(E)x \ // x \text{ is a global synchronizer} \\
& \mid ?x@q \mid ?x@(E) \mid bx@q \mid bx@(E) \\
C ::= & \text{ACT} \mid \{ C \} \mid C \ C \mid \text{if } (B) \ C \ \text{else } C \mid \text{while } (B) \ C \\
\text{ACT} ::= & \ ; \mid \text{goto } name \ ; \mid x = E \ ; \\
\end{align*}
\]

Guarded commands with modes (GCM) - a language for state-transition graphs

\[
\begin{align*}
\text{INI} ::= & \text{initially } B; \\
\text{SP} ::= & \text{RTASK } B; \mid \text{tctl } T; \mid \text{GTASK } GS ; \ GS ; \\
\text{RTASK} ::= & \text{safety} \mid \text{goal} \mid \text{risk} \\
T ::= & B \mid (T) \mid \text{forall always } K T \mid \text{exists always } K T \\
& \mid \text{forall eventually } K T \mid \text{exists eventually } K T \\
& \mid \text{forall } T \ \text{until } K T \mid \text{exists } T \ \text{until } K T \\
& \mid \text{forall } x : c .. c, T \mid \text{exists } x : c .. c, T \\
K ::= & \{(c, c)\} \mid \{(c, c)\} \mid \{(c, D)\} \mid \{(c, D)\} \\
D ::= & c \mid oo
\end{align*}
\]
Guarded commands with modes (GCM) - a language for state-transition graphs

GTASK ::= check branching simulation
       | check branching bisimulation
GS ::= c | c, GS

A state-transition - represented as a GCM

a1 ➔ (true) ➔ w = 0;
   ➔ (true) ➔ x = 0;
   ➔ (true) ➔ y = z^2;
   ➔ (x >= y) ➔;
   ➔ (x < y) ➔;
   ➔ (true) ➔ if(w > z^2*z)*w = z^2*z;
   ➔ a0

a2

a3

a4

a5

a6
A state-transition
- represented as a GCM

process count = 1;
global discrete w,x,y,z:0..5;
mode a1 (true) { when (true) may w = 0; goto a2; }
mode a2 (true) { when (true) may x = 0; goto a3; }
mode a3 (true) { when (true) may y = z*z; goto a4; }
mode a4 (true) { when (x>=y) may goto a8;
    when (x < y) may goto a5;
}
mode a5 (true) { when (true) may w=w+x*z; goto a6; }
mode a6 (true) { when (true) may x=x+1; goto a4; }
mode a8 (true) { when (true) may if (w>z*z*z) w= z*z*z; }
initially a1[1]&&w==1&&x==1&&y==1&&z==1;

A state-transition
- represented as a GCM

process count = 1;
global discrete w,x,y,z:0..5;
mode a1.2 (true) { when (true) may w = 0; x = 0; goto a3; }
mode a3 (true) { when (true) may y = z*z; goto a4; }
mode a4 (true) { when (x>=y) may goto a8;
    when (x < y) may goto a5;
}
mode a5 (true) { when (true) may w=w+x*z; goto a6; }
mode a6 (true) { when (true) may x=x+1; goto a4; }
mode a8 (true) { when (true) may if (w>z*z*z) w= z*z*z; }
initially a1[1]&&w==1&&x==1&&y==1&&z==1;
Guarded commands with modes (GCM)

1: w = 0;
2: x = 0;
3: y = z*z;
4: while (x < y) {
5:   w = w + x*z;
6:   x = x + 1;
7: }
8: if (w > z*z*z) w = z*z*z;

Guarded commands
when (pc==1) may w = 0; pc=2;
when (pc==2) may x = 0; pc=3;
when (pc==3) may y = z*z; pc=4;
when (pc==4&&x>=y) may pc=8;
when (pc==4&&x < y) may pc=5;
when (pc==5) may w=w+x*z; pc=6;
when (pc==6) may x=x+1; pc=4;
when (pc==8) may if (w>z*z*z) w= z*z*z;

Concurrent programs

- A set programs running independently, communicating from time to time, thereby performing a common task.
- Flavors of Concurrency
  - Synchronous execution
  - Asynchronous / interleaved execution
    - Communication via shared variables
    - Message passing communication
**Kripke Structure**
- for a concurrent system

- Programs (as opposed to circuits) are typically considered asynchronous.
- An asynchronous concurrent system is a collection of sequential programs $P_1...P_k$ running in parallel with only one pgm. making a move at every time step.
  - How do the sequential programs communicate?
  - What are the behaviors of the concurrent system?

---

**Kripke Structure**
- for a concurrent system

- Behaviors of each sequential program $P_i$ captured by its operational semantic.
- The programs $P_i$ need not be terminating.
- Behaviors (Traces) of $P_1...P_k$ formed by *interleaving* the transitions of the programs.
- Consider two non-communicating programs.
Guarded commands - for a concurrent system
Interleavings

Semantics as
Kripke structure

Guarded commands - for a concurrent system
Interleavings

Semantics as
Kripke structure
Kripke Structure
- for a concurrent system

- Obtaining Kripke Structure from a concurrent program directly is laborious.
- Typically, model checking tools allow you to input the program in its modeling language, and then it extracts the Kripke Structure (or some succinct version of it).
- Model the sequential pgms. separately and specify a model of concurrency
e.g. asynchronous with shared variable communication
Kripke Structure
- A Mutual Exclusion Example

// 2 processes that communicate with a shared variable.
process count = 2;
global discrete turn: 0..1;

// state-transition graph for process 1
mode a0 (true) { when (turn==0) may goto a1; }
mode a1 (true) { when (true) may turn = 1; goto a0; }

// state-transition graph for process 2
mode b0 (true) { when (turn==1) may goto b1; }
mode b1 (true) { when (true) may turn = 0; goto b0; }

initially a0[1] && b0[2];

---

Kripke Structure
- for a concurrent system states

states can be recorded as

(mode of 1, mode of 2, value of turn)

- mode of 1 \(\in\) {a0, a1}
- mode of 2 \(\in\) {b0, b1}
- The value of turn \(\in\) {0, 1}
- There are 8 states.
- Not all of them are reachable from the initial state.
State-transition graphs

- Synchronization

The reader process
(1) buf = 0;
(2) while true, do {
(3) if (buf == 0), read;
(4) buf = 0;
(5) }

The writer process
(6) d = 1;
(7) while true, do {
(8) if (buf != 0),
(9) write buf = d;d=0;
(10) d = 1;
(11) }
State-transition graphs
- Synchronization
Kripke structure (part)

State-transition graphs
- Semantics of concurrency (I)
Interleaving semantics
(the yellow arcs).
- At most one autonomous
  party may execute at a time.
State-transition graphs
- Semantics of concurrency (I ...)

Interleaving semantics (the yellow arcs).

- At most one autonomous party may execute at a time.

E.g.,

State-transition graphs
- Semantics of concurrency (II)

Synchronizations (the red arc)

- All communicating parties in a minimal & autonomous synchronization must execute at the same time.
State-transition graphs
- Semantics of concurrency (II)

Synchronizations (the red arc)
- All communicating parties in a minimal & autonomous synchronization must execute at the same time.

E.g.,

State-transition graphs

CSMA/CD protocol, the Ethernet protocol
- 500m in expanse
- 2500m in expanse with repeaters
  - Round-trip 48 µs.
- Messages length at least 64 bytes to detect round-trip corruption.
State-transition graphs
- CSMA/CD

State-transition graph for automata
- an exercise

Please construct an automata with
- input alphabet \{1,0,e\}
- output alphabet \{1,0\}
- reads in eeb_{n}b_{n-1}…b_{1}b_{0}
- output 3^*(b_{n}b_{n-1}…b_{1}b_{0}) with b_{n} as the most significant bit.

Example:
when input is ee1011(11), output is 100001(33)
 ee11(3), 1001(9)
State-transition graph for automata - an exercise

State-transition graph for automata - an exercise run for ee11(3)
State-transition graph for automata - an exercise run for ee11(3)

State-transition graph for automata - an exercise run for ee11(3)
State-transition graph for automata
- an exercise run for ee11(3)

State-transition graph for automata
- an exercise run for ee11(3)
State-transition graph for automata - an exercise run for ee1011(11)

ee1011(11)
State-transition graph for automata - an exercise run for ee1011(11)

State-transition graph for automata - an exercise run for ee1011(11)
State-transition graph for automata - an exercise run for ee1011(11)

State-transition graph for automata - an exercise run for ee1011(11)
State-transition graph for automata - an exercise run for ee1011(11)

\[
a_{n+2}a_{n+1}...a_1a_0 = 3(eeb_n b_{n-1}...b_1b_0) \\
\Rightarrow a_{k+1} = b_{k+1} + b_k + c_k
\]
State-transition graph for automata

- How to construct an automata for \( c \in \mathbb{N} \),
  \( c^*(b_n b_{n-1} \ldots b_1 b_0) \)
  Need \( \lceil \log_2(c) \rceil \times 2^{\lceil \log_2(c) \rceil} + 1 \) states!
- How to construct an automata for
  \( a_n a_{n-1} \ldots a_1 a_0 + b_n b_{n-1} \ldots b_1 b_0 \)?
  Can you do this?
- How to construct an automata for
  \( \sum_{c_k} c_k^*(b_{n,k} b_{n-1,k} \ldots b_{1,k} b_{0,k}) \)?

Kripke Structures
- composition for a concurrent system

Given \( A_i = \langle S_i, S_{i,0}, R_i, L_i \rangle, 1 \leq i \leq n \)

Cartesian Product of \( A_1, A_2, \ldots, A_n \),
\( A = \langle S, S_0, R, L \rangle \)

- \( S : S_1 \times S_2 \times \ldots \times S_n \)
- \( S_0 : S_{1,0} \times S_{2,0} \times \ldots \times S_{n,0} \)
- \( R([s_1, \ldots, s_{j-1}, s_j, s_{j+1}, \ldots, s_n], [s_1, \ldots, s_{j-1}, s'_{j}, s_{j+1}, \ldots, s_n]) \)
  - \((s_i, s'_i) \in R_i \)
  - According to the interleaving semantics, one process transition at a moment

- \( L([s_1, s_2, \ldots, s_n]) = L_1(s_1) \cup L_2(s_2) \cup \ldots \cup L_n(s_n) \)
Kripke Structures
- Cartesian product method
1. Construct all the vectors of component process states
2. Eliminate all those inconsistent vectors according to invariance condition
3. Draw arcs from vectors to vectors according to process transitions
   - Very often creates many unreachable states

Kripke structure
- Practical algorithm for construction
Given \( A=\langle S, S_0, R, L \rangle \)
- Usually only \( S_0, R, L \) are given.
- We may want to construct \( S \).
- Usually \( S \) is too big to construct.
Kripke Structures
- on-the-fly method
1. Starting from the initial states (or goal states in backward analysis)
2. Step by step, add states that is reachable from those already reached, until no more new reachable states are generated.
   - *Tedious but may result in much smaller reachable state-space representation.*

Kripke Structures
- forward reachability analysis
- Use strongest postcondition to compute state-spaces forward reachable from initial states
- Can only be used for safety analysis
- Very often can lead to larger state-space representation
- Very often can lead to unnecessary total ordering enumeration
  - Need symmetry reduction and partial-order reduction
Kripke Structures
- backward reachability analysis

- Use weakest precondition to compute state-spaces backward reachable from goal states
- The mandatory method for model-checking
- More like refutation
- Very often can lead to smaller state-space representation
- Very often can lead to less total ordering enumeration

Kripke Structure
- propositions

Given by the valuation of the variables defining the states. Possible propositions

\[ pc_0 = l_0, \ldots, pc_0 = l_3 \]
\[ pc_1 = m_0, \ldots, pc_1 = m_3 \]
\[ turn = 0, turn = 1 \]

Clearly the proposition \( pc_0 = l_0 \) is true in any state of the form \( \langle pc_0 = l_0, pc_1 = ?, turn = ? \rangle \)

This clarifies the labeling function \( L \) in Kripke Structure
**Kripke Structure**
- system properties

- Propositions can be combined to state interesting properties
  - It is never the case that \( pc_0 = l_2 \) and \( pc_1 = m_2 \)
  - The above is the mutual exclusion property.
  - We will study a logic for describing properties in next class.

---

**Kripke Structure**
- fairness in a concurrent system

In a concurrent system, there could be several independent modules with independent descriptions.

- How can we construct the Kripke structure for global behavior description?
- How can we run the modules *fairly*?
  - Is there a module that never gets execution in interleaving semantics?
  - Is an unfair execution meaningless?
Fairness in concurrent systems

Semantics as

Kripke structure

state-transition graphs

Kripke Structure
- fairness in a concurrent system

- Proc0 manipulates X
- Proc1 manipulates Y
- In the global state \( <X=0, Y=1> \)
  - Proc0 or Proc1 could make a move.
  - We allow the behavior that Proc1 always makes a move (self-loop)
  - System is stuck at \( <X=0, Y=1> \)
  - Unfair execution!
Fair Kripke Structures

- \( M = (S, S_0, R, L, F) \)
  - \( S, S_0, R, L \) as before.
  - \( F \subseteq 2^S \) is a set of fairness constraints.
  - Each element of \( F \) is a set of states which must occur infinitely often in any execution path.
- In our example, \( F = \{\langle X=1, Y=1 \rangle \} \)
  - Avoid getting stuck at \( \langle X=0, Y=1 \rangle \) or \( \langle X=1, Y=0 \rangle \)

Kripke structure - verification

- safety analysis
  - Can the system be always safe?
  - Can a risk state happen?
- liveness analysis
  - Can the job be done sometimes?
  - Can the job be prevented from been done?
- bisimulation checking
  - Are two Kripke structures the same transition by transition?
- simulation checking
  - Can one Kripke structure match every transition by the another?
- language inclusion
  - Are all traces of one Kripke structure also ones of another?
Model-checking frameworks in our lecture

F: set of fairness assumptions.
✓: known;
☑: discussed in the lecture

<table>
<thead>
<tr>
<th>Spec</th>
<th>Logics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>traces</td>
</tr>
<tr>
<td>Model</td>
<td></td>
</tr>
<tr>
<td>F=Ø</td>
<td>✓</td>
</tr>
<tr>
<td>F≠Ø</td>
<td>✓</td>
</tr>
<tr>
<td>Trees</td>
<td>☑</td>
</tr>
<tr>
<td>F=Ø</td>
<td>☑</td>
</tr>
<tr>
<td>F≠Ø</td>
<td>☑</td>
</tr>
<tr>
<td>Linear</td>
<td>☑</td>
</tr>
<tr>
<td>F=Ø</td>
<td>☑</td>
</tr>
<tr>
<td>F≠Ø</td>
<td>☑</td>
</tr>
<tr>
<td>Branching</td>
<td>☑</td>
</tr>
<tr>
<td>F=Ø</td>
<td>☑</td>
</tr>
<tr>
<td>F≠Ø</td>
<td>☑</td>
</tr>
</tbody>
</table>

Kripke structure
- safety analysis

Given
- a Kripke structure $A = (S, S_0, R, L)$
- a safety predicate $\eta$

Can $\eta$ be false at some state along some runs?

Example:
Can the engine stall?
Can the boiler be overheated?

$\neg(\text{PC}_0=\text{CR}_0 \land \text{PC}_1=\text{CR}_1)$ is an invariant!
Kripke structure - safety analysis

Reachability algorithm in graph theory
Given
- a Kripke structure \( A = (S, S_0, R, L) \)
- a safety predicate \( \eta \),

find a path from a state in \( S_0 \) to a state in \([¬\eta]\).

Solutions in graph theory
- Shortest distance algorithms
- spanning tree algorithms

/* Given \( A = (S, S_0, R, L) \) */
safety_analysis(\( \eta \)) /* using least fixpoint algorithm */ {
    for all \( s \), if \( \eta \in L(s) \), \( L(s) = L(s) \cup \{\exists \diamond ¬\eta\} \);
    repeat {
        for all \( s \), if \( \exists (s,s') (\exists \diamond ¬\eta \in L(s')) \),
        \( L(s) = L(s) \cup \{\exists \diamond ¬\eta\} \);
    } until no more changes to \( L(s) \) for any \( s \).
    if there is an \( s_0 \in S_0 \) with \( \exists \diamond ¬\eta \in L(s_0) \), return ‘unsafe,’
    else return ‘safe.’
}

The procedure terminates since \( S \) is finite in the Kripke structure.
Kripke structure
- safety analysis

Least fixpoint in modal logics
iterative expansion

Dark-night murder, strategy I:
A suspect will be in the 2nd round iff

- He/she lied to the police in the 1st round; or
- He/she is loyal to someone in the 2nd round

What is the minimal solution to $2nd[i]$?

$Liar[i] \lor \exists j \neq i (2nd[j] \land Loyal\text{-}to[i,j]) \rightarrow 2nd[i]$
Kripke structure
- **Least fixpoint in modal logics**

In a dark night, there was a cruel murder.

- n suspects, numbered 0 through n-1.
- $Liar[i]$ iff suspect i has lied to the police in the 1st round investigation.
- $Loyal-to[i,j]$ iff suspect i is loyal to suspect j in the same criminal gang.
- $2nd[i]$ iff suspect i to be in 2nd round investigation.

What is the minimal solution to $2nd[]$?

---

Kripke structure
- **Greatest fixpoint in modal logics**

In a dark night, there was a cruel murder.

- n suspects, numbered 0 through n-1.
- $\neg Liar[i]$ iff the police cannot prove suspect i has lied to the police in the 1st round investigation.
- $Loyal-to[i,j]$ iff suspect i is loyal to j and j is not in the 2nd round.
- $2nd[i]$ iff suspect i to be in 2nd round investigation.

What is the maximal solution to $\neg 2nd[]$?
**Kripke structure**

- Greatest fixpoint in modal logics

Dark-night murder, strategy II
A suspect will not be in the 2nd round iff

- We cannot prove he/she has lied to the police; and
- He/she is loyal to someone not in the 2nd round.

What is the maximal solution to $\neg 2nd[i]$?

$\neg 2nd[i] \equiv \neg Liar[i] \land \exists j (\neg 2nd[j] \land Loyal-to[i,j])$

In comparison:

$\neg 2nd[i] \equiv \neg Liar[i] \land \forall j (\neg 2nd[j] \land Loyal-to[i,j])$

$\neg 2nd[i] \equiv \neg Liar[i] \land \forall j (\neg 2nd[j] \Rightarrow Loyal-to[i,j])$

$\neg 2nd[i] \equiv \neg Liar[i] \land \forall j (Loyal-to[i,j] \Rightarrow \neg 2nd[j])$

---

**CTL**

- Symbolic model-checking with BDD

- In a Kripke structure, states are described with binary variables.

  $\text{n binary variables} \rightarrow 2^n \text{ states}$

  $x_1, x_2, \ldots, x_n$

- We can use a BDD to describe legal states.

  A Boolean function with $n$ binary variables

  $S(x_1, x_2, \ldots, x_n)$
**CTL - symbolic model-checking with Propositional logics**

*Example:*

$x_1 \ x_2 \ x_3$

\[
S(x_1, x_2, x_3) = (x_1 \land \neg x_2 \land x_3) \\
\lor (\neg x_1 \land \neg x_2 \land x_3) \\
\lor (\neg x_1 \land x_2 \land \neg x_3)
\]

---

**CTL - symbolic model-checking with Propositional logics**

State transition relation as a logic function with $2n$ parameters

\[
R(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)
\]
**CTL - symbolic model-checking with Propositional logics**

\[ x_1 \ x_2 \ x_3 \ x'_1 \ x'_2 \ x'_3 \]

\[ \begin{array}{c}
101 \\
\rightarrow \\
001 \\
\rightarrow \\
010 \\
\end{array} \]

\[
R(x_1, x_2, x_3, x'_1, x'_2, x'_3) = \\
(x_1 \land \neg x_2 \land x_3 \land \neg x'_1 \land x'_2 \land x'_3) \\
\lor \\
(x'_1 \land \neg x_2 \land x_3 \land x'_1 \land x'_2 \land \neg x'_3) \\
\lor \\
(x'_1 \land x_2 \land \neg x_3 \land \neg x'_1 \land \neg x'_2 \land \neg x'_3)
\]

---

**CTL - symbolic model-checking with Propositional logics**

Path relation also as a logic function with \(2n\) parameters

\[
\text{reach}(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)
\]
CTL - symbolic model-checking with Propositional logics

\[ x_1 \ x_2 \ x_3 \ x'_1 \ x'_2 \ x'_3 \]

reach\((x_1, x_2, x_3, x'_1, x'_2, x'_3)\) =
\[
\begin{align*}
(x_1 \land \neg x_2 \land x_3 \land \neg x'_1 \land \neg x'_2 \land x'_3) \\
\lor (x_1 \land \neg x_2 \land x_3 \land \neg x'_1 \land x'_2 \land \neg x'_3) \\
\lor (\neg x_1 \land \neg x_2 \land x_3 \land \neg x'_1 \land x'_2 \land \neg x'_3) \\
\lor (\neg x_1 \land x_2 \land \neg x_3 \land \neg x'_1 \land \neg x'_2 \land x'_3) \\
\lor (\neg x_1 \land \neg x_2 \land x_3 \land \neg x'_1 \land \neg x'_2 \land x'_3) \\
\lor (\neg x_1 \land x_2 \land \neg x_3 \land \neg x'_1 \land \neg x'_2 \land \neg x'_3)
\end{align*}
\]

Symbolic safety analysis

- \(I\) : initial condition with parameters
  \[x, x_2, \ldots, x_n\]
- \(\eta\) : safe condition with parameters
  \[x_1, x_2, \ldots, x_n\]

If \(I \land \neg(\eta \uparrow) \land \text{reach}(x_1, x_2, \ldots, x_m, x'_1, x'_2, \ldots, x'_n)\) is not false,
- a risk state is reachable.
- \textbf{the system is not safe.}
Symbolic safety analysis
- construction of reach\((x_1, \ldots, x_n, x'_1, \ldots, x'_n)\)

\[
R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)
\land \exists y_1, \ldots, \exists y_n \left( \neg \neg R(x_1, \ldots, x_n, y_1, \ldots, y_n)
\land \reach(y_1, \ldots, y_n, x'_1, \ldots, x'_n) \right)
\]

\[
\neg \neg \reach(x_1, \ldots, x_n, x'_1, \ldots, x'_n)
\]

This is a least fixpoint for backward analysis.

Symbolic safety analysis
- construction of reach\((x_1, \ldots, x_n, x'_1, \ldots, x'_n)\)

\[
R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)
\land \exists y_1, \ldots, \exists y_n \left( \reach(x_1, \ldots, x_n, y_1, \ldots, y_n)
\land \reach(y_1, \ldots, y_n, x'_1, \ldots, x'_n) \right)
\]

\[
\neg \neg \reach(x_1, \ldots, x_n, x'_1, \ldots, x'_n)
\]

This is another least fixpoint for speed-up.
Symbolic safety analysis
- construction of reach($x_1, \ldots, x_n, x'_1, \ldots, x'_n$)

$R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$
$\forall \exists y_1, \ldots, \exists y_n (reach(x_1, \ldots, x_n, y_1, \ldots, y_n)$
$\wedge R(y_1, \ldots, y_n, x'_1, \ldots, x'_n)$
$\Rightarrow$ reach($x_1, \ldots, x_n, x'_1, \ldots, x'_n$)
This is another least fixpoint for forward analysis.

Symbolic safety analysis (backward)

Encode the states with variables $x_0, x_1, \ldots, x_n$.
- the state set as a proposition formula: $S(x_0, x_1, \ldots, x_n)$
- the risk state set as $\neg \eta(x_0, x_1, \ldots, x_n)$
- the initial state set as $I(x_0, x_1, \ldots, x_n)$
- the transition set as $R(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n)$

$b_0 = \neg \eta(x_0, x_1, \ldots, x_n) \land S(x_0, x_1, \ldots, x_n); k = 1$
repeat
  $b_k = b_{k-1} \lor \exists x'_0 \exists x'_1 \ldots \exists x'_n (R(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n) \land (b_{k-1} \uparrow))$;
  $k = k + 1$;
until $b_k \equiv b_{k-1}$;
if ($b_k \land I(x_0, x_1, \ldots, x_n)) \equiv \text{false}$, return ‘safe’; else return ‘risky’;
**Kripke structure - symbolic safety analysis**

- **states:** \( S(x,y,z) \equiv (\neg x \land \neg y \land \neg z) \lor (\neg x \land \neg y \land z) \lor (\neg x \land y \land \neg z) \lor (\neg x \land y \land z) \lor (\neg x \lor (x \land \neg y)) \)
- **initial state:** \( I(x,y,z) \equiv \neg x \land \neg y \land \neg z \)
- **risk state:** \( \neg \eta(x,y,z) \equiv x \land \neg y \land z \)

**Transitions:**

\[
R(x,y,z,x',y',z') \equiv \\
(\neg x \land \neg y \land \neg z \land \neg x' \land \neg y' \land z') \lor (\neg x \land y \land \neg z \land \neg x' \land y' \land z') \lor (\neg x \land y \land z \land \neg x' \land y' \land \neg z') \lor (\neg x \land y \land z \land x' \land \neg y' \land \neg z') \lor (\neg x \land y \land z \land x' \land y' \land \neg z') \lor (\neg x \land y \land z \land x' \land y' \land z')
\]
Symbolic safety analysis (backward)

\[ b_0 = \neg \eta(x,y,z) \equiv x \land \neg y \land \neg z \]

\[ b_1 = b_0 \lor \exists x' \exists y' \exists z' (R(x,y,z,x',y',z') \land b_0 \uparrow) \]
\[ = (x \land \neg y \land \neg z) \lor \exists x' \exists y' \exists z' (R(x,y,z,x',y',z') \land x' \land \neg y' \land \neg z') \]
\[ = (x \land \neg y \land \neg z) \lor \exists x' \exists y' \exists z' (((\neg x \land x \land y \land \neg z) \lor (\neg x \land y \land z)) \land x' \land \neg y' \land \neg z') \]
\[ = (x \land \neg y \land \neg z) \lor (\neg x \land x \land y \land z) \lor (\neg x \land y \land z) \]

\[ b_2 = b_1 \lor \exists x' \exists y' \exists z' (R(x,y,z,x',y',z') \land b_1 \uparrow) \]
\[ = (\neg x \lor \neg x \land y \land z) \lor (\neg x \land \neg x \land y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land y \land z) \]
\[ b_3 = b_2 \lor \exists x' \exists y' \exists z' (R(x,y,z,x',y',z') \land b_2 \uparrow) \]
\[ = (\neg x \lor \neg x \land y \land z) \lor (\neg x \land \neg x \land y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land y \land z) \]
\[ b_4 = b_3 \lor \exists x' \exists y' \exists z' (R(x,y,z,x',y',z') \land b_3 \uparrow) \]
\[ = (\neg x \lor \neg x \land y \land z) \lor (\neg x \land \neg x \land y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land y \land z) \]
\[ b_4 \land I(x,y,z) = (\neg x \land \neg y \land \neg z) \]

fixpoint
non-empty intersection with the initial condition
risk detected.
老子道德經四十一章

大音希聲
大象無形
道隱無名

Symbolic safety analysis (backward)

One assumption for the correctness!

- Two states cannot be with the same proposition labeling.
- Otherwise, the collapsing of the states may cause problem.

may need a few propositions for the names of the states.
Symbolic safety analysis (forward)

Encode the states with variables $x_0, x_1, \ldots, x_n$.

- the state set as a proposition formula: $S(x_0, x_1, \ldots, x_n)$
- the risk state set as $\neg \eta(x_0, x_1, \ldots, x_n)$
- the initial state set as $I(x_0, x_1, \ldots, x_n)$
- the transition set as $R(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n)$

$f_0 = I(x_0, x_1, \ldots, x_n) \land S(x_0, x_1, \ldots, x_n)$; $k = 1$

repeat

$f_k = f_{k-1} \lor (\exists x_0 \exists x_1 \ldots \exists x_n (R(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n) \land f_{k-1}))$ ↓

$k = k + 1$

until $f_k \equiv f_{k-1}$

if ($f_k \land \neg \eta(x_0, x_1, \ldots, x_n)$) \equiv false, return ‘safe’; else return ‘risky’;

Symbolic safety analysis (forward)

$f_0 = I(x,y,z) \equiv \neg x \land \neg y \land \neg z$

$f_1 = f_0 \lor (\exists x \exists y \exists z (R(x,y,z,x',y',z') \land f_0))$ ↓

$= (-x \land \neg y \land \neg z) \lor (\exists x \exists y \exists z (R(x,y,z,x',y',z') \land \neg x \land \neg y \land \neg z))$ ↓

$= (-x \land \neg y \land \neg z) \lor (\exists x \exists y \exists z (\neg x' \land \neg y' \land z' \land \neg x \land \neg y \land \neg z))$ ↓

$= (-x \land \neg y \land \neg z) \lor (\neg x' \land \neg y \land z')$ ↓

$= (-x \land \neg y \land \neg z) \lor (x \land \neg y \land z) = -x \land -y$

$f_2 = f_1 \lor (\exists x \exists y \exists z (R(x,y,z,x',y',z') \land f_1))$ ↓

$= (-x \land \neg y \land \neg z) \lor (\neg x \land \neg y \land \neg z)$

$f_3 = f_2 \lor (\exists x \exists y \exists z (R(x,y,z,x',y',z') \land f_2))$ ↓

$= (-y) \lor (\neg x \land \neg y \land \neg z)$

$f_4 = f_3 \lor (\exists x \exists y \exists z (R(x,y,z,x',y',z') \land f_3))$ ↓

$= (-y) \lor (\neg x \land \neg y \land \neg z)$

$f_4 \land \neg \eta(x,y,z) = ((\neg y) \lor (\neg x \land \neg y \land \neg z)) \land (x \land \neg y \land \neg z) = (x \land \neg y \land \neg z)$

fixpoint

non-empty intersection with the risk condition → risk detected.
Bounded model-checking

Encode the states with variables $x_{0,k}, x_{1,k}, \ldots, x_{n,k}$.

- the state set as a proposition formula: $S(x_{0,k}, x_{1,k}, \ldots, x_{n,k})$
- the risk state set as $\neg \eta(x_{0,k}, x_{1,k}, \ldots, x_{n,k})$
- the initial state set as $I(x_{0,0}, x_{1,0}, \ldots, x_{n,0})$
- the transition set as $R(x_{0,k-1}, x_{1,k-1}, \ldots, x_{n,k-1}, x_{0,k}, x_{1,k}, \ldots, x_{n,k})$

$f_0 = I(x_{0,0}, x_{1,0}, \ldots, x_{n,0}) \land S(x_{0,0}, x_{1,0}, \ldots, x_{n,0}); k = 1$

repeat

$f_k = R(x_{0,k-1}, x_{1,k-1}, \ldots, x_{n,k-1}, x_{0,k}, x_{1,k}, \ldots, x_{n,k}) \land f_{k-1}$

$k = k + 1$

until $f_k \land \neg \eta(x_{0,k}, x_{1,k}, \ldots, x_{n,k}) \neq \text{false}$

When to stop?
1. diameter of the state graph
2. explosion up to tens of steps.

Bounded model-checking

\[
\begin{align*}
f_0 &= I(x, y, z) \equiv \neg x \land \neg y \land \neg z \\
f_1 &= R(x_0, y_0, z_0, x_1, y_1, z_1) \land f_0 \equiv \neg x_0 \land \neg y_0 \land \neg z_0 \land \neg x_1 \land \neg y_1 \land z_1 \\
f_2 &= R(x_1, y_1, z_1, x_2, y_2, z_2) \land f_1 \\
&= \neg x_0 \land \neg y_0 \land \neg z_0 \land \neg x_1 \land \neg y_1 \land z_1 \land (\neg x_2 \land \neg y_2 \land \neg z_2) \lor (\neg x_2 \land y_2 \land \neg z_2) \\
f_3 &= R(x_2, y_2, z_2, x_3, y_3, z_3) \land f_2 \\
&= \neg x_0 \land \neg y_0 \land \neg z_0 \land \neg x_1 \land \neg y_1 \land z_1 \\
&\land (\neg x_2 \land \neg y_2 \land \neg z_2 \land \neg x_3 \land \neg y_3 \land \neg z_3) \\
&\land (\neg x_2 \land y_2 \land \neg z_2 \land (x_3 \land \neg y_3 \land \neg z_3) \lor (x_3 \land y_3 \land \neg z_3)) \\
&\land (\neg x_3 \land \neg y_3 \land \neg z_3) \\
f_3 \land \neg \eta(x_3, y_3, z_3) &= (x_3 \land \neg y_3 \land \neg z_3)
\end{align*}
\]
Transition relation from state-transition graphs

Given a set of rules \( r_1, r_2, \ldots, r_m \) of the form
\[ r_k: \text{when } (\tau_k) \text{ may } y_{k,0}=d_0; y_{k,1}=d_1; \ldots; y_{k,n_k}=d_{n_k}; \]

\[ R(x_0,x_1,\ldots,x_n,x'_0,x'_1,\ldots,x'_n) \]
\[ \equiv \bigvee_{k \in [1,m]} \left( \tau_k \land y'_{k,0}=d_0 \land y'_{k,1}=d_1 \land \ldots \land y'_{k,n_k}=d_{n_k} \right) \]
\[ \land \bigwedge_{h \in [1,n]} \left( x_h \notin \{y_{k,0}, y_{k,1}, \ldots, y_{k,n_k}\} \Rightarrow x'_h = x'_h \right) \]

Transition relation from GCM rules.

Given a set of rules for \( X=\{x,y,z\} \)
\[ r_1: \text{when } (x<y && y>2) \text{ may } y=x+y; x=3; \]
\[ r_2: \text{when } (z>=2) \text{ may } y=x+1; z=0; \]
\[ r_3: \text{when } (x<2) \text{ may } x=0; \]

\[ R(x_0,x_1,\ldots,x_n,x'_0,x'_1,\ldots,x'_n) \]
\[ \equiv (x<y \land y>2 \land y'=x+y \land x'=3 \land z'=z) \]
\[ \lor (z>=2 \land y'=x+1 \land z'=0 \land x'=x) \]
\[ \lor (x<2 \land x'=0 \land y'=y \land z'=z) \]
Transition relation
- from state-transition graphs
In general, transition relation is expensive to construct.
Can we do the following state-space construction
\[ \exists x'_0 \exists x'_1 ... \exists x'_n (R(x_0,x_1,...,x_n,x'_0,x'_1,...,x'_n) \land (b_{k-1} \uparrow)) \]
directly with the GCM rules?

Yes, **on-the-fly state space construction.**

---

On-the-fly precondition calculation with GCM rules.

\[ \exists x'_0 \exists x'_1 ... \exists x'_n (R(x_0,x_1,...,x_n,x'_0,x'_1,...,x'_n) \land (b \uparrow)) \]
\[ \equiv \forall_{k \in [1,m]} (\tau_k \land \exists y_{k,0} \exists y_{k,1} ... \exists y_{k,nk} (b \land \land_{h \in [0,nk]} y_{k,h} = d_h)) \]

pre(b) {
    r = false;
    for k = 1 to m, {
        let f = b;
        for h=nk to 0, f = \exists y_{k,h} (f \land y_{k,h} = d_h);
        r = r \lor (\tau_k \land f);
    }
    return (r);
}
On-the-fly precondition calculation with GCM rules.

Given a set of rules \( r_1, r_2, \ldots, r_m \) of the form
\[
r_k: \text{when } (\tau_k) \text{ may } y_{k,0}=d_0; \ y_{k,1}=d_1; \ \ldots; \ y_{k,n_k}=d_{n_k};
\]
\[
\exists x'_0 \exists x'_1 \ldots \exists x'_n (R(x_0,x_1,\ldots,x_n,x'_0,x'_1,\ldots,x'_n) \land (b \uparrow))
\]
\[
= \bigvee_{k \in [1,m]} \left( \tau_k \land \exists y_{k,0} \exists y_{k,1} \ldots \exists y_{k,n_k} \left( b \land \bigwedge_{h \in [0,n_k]} y_{k,h}=d_h \right) \right)
\]

However, GCM rules are more complex than that.

On-the-fly precondition calculation with GCM rules.

Given a set of rules for \( X=\{x,y,z\} \)
\[
r_1: \text{when } (x<y \&\& y>2) \text{ may } y=z; \ x=3;
\]
\[
r_2: \text{when } (z>=2) \text{ may } y=x+1; \ z=7;
\]
\[
r_3: \text{when } (x<2) \text{ may } z=0;
\]
\[
\exists x'_0 \exists x'_1 \ldots \exists x'_n (R(x_0,x_1,\ldots,x_n,x'_0,x'_1,\ldots,x'_n) \land (x<4 \land z>5) \uparrow)
\]
\[
= (x<y \land y>2 \land \exists y \exists x( x<4 \land z>5 \land y==z \land x==3))
\]
\[
\lor (z>=2 \land \exists y \exists z( x<4 \land z>5 \land y==x+1 \land z==7))
\]
\[
\lor (x<2 \land \exists z( x<4 \land z>5 \land z==0))
\]
\[
= (x<y \land y>2 \land z>5) \lor (z>=2 \land x<4) \lor (x<2 \land \exists z(\text{false}))
\]
\[
= (x<y \land y>2 \land z>5) \lor (z>=2 \land x<4)
\]
On-the-fly precondition calculation with GCM rules.

Given a set of rules \( r_1, r_2, \ldots, r_m \) of the form

\[ r_k: \text{when } (\tau_k) \text{ may } s_k; \]

\[
\exists x'_0 \exists x'_1 \ldots \exists x'_n (t(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n) \land (b \uparrow)) = \bigvee_{k \in [1,m]} (\tau_k \land \text{pre}(s_k, b))
\]

What is \( \text{pre}(s, b) \)?

A general propositional formula

precondition procedure

A GCM statement

---

On-the-fly precondition calculation with GCM rules.

Given a set of rules \( r_1, r_2, \ldots, r_m \) of the form

\[ r_k: \text{when } (\tau_k) \text{ may } s_k; \]

What is \( \text{pre}(s, b) \)?

\[ \text{pre}(x = E; b) \equiv b[x/E] \]

- Ex 1. the precondition to \( x=x+z; \)
  \( (x==y+2 \land x<4 \land z>5) \equiv x+z==y+2 \land x+z<4 \land z>5 \)

- Ex 2. the precondition to \( x=5; \)
  \( (x==y+2 \land x<4 \land z>5) \equiv 5==y+2 \land 5<4 \land z>5 \)

- Ex 3. the precondition to \( x=2x+1; \)
  \( (x==y+2 \land x<4 \land z>5) \equiv 2^x+1==y+2 \land 2^x+1<4 \land z>5 \)
On-the-fly precondition calculation with GCM rules.

Given a set of rules \( r_1, r_2, \ldots, r_m \) of the form

\[
r_k: \text{when } (\tau_k) \text{ may } s_k;
\]

What is \( \text{pre}(s, b) \)?

- \( \text{pre}(x = E; b) \equiv b[x/E] \)
- \( \text{pre}(s_1 s_2, b) \equiv \text{pre}(s_1, \text{pre}(s_2, b)) \)
- \( \text{pre}(\text{if } (B) s_1 \text{else } s_2) \equiv (B \land \text{pre}(s_1, b)) \lor (\neg B \land \text{pre}(s_2, b)) \)
- \( \text{pre}(\text{while } (B) s, b) \equiv \ldots \)

new expression obtained from \( b \) by replacing every occurrence of \( x \) with \( E \).

Ex. the precondition to \( x = x + z; \)

\[
(x==y+2 \land x<4 \land z>5) \ [x/x+z] = x+z==y+2 \land x+z<4 \land z>5
\]

On-the-fly precondition calculation with GCM rules.

Given a set of rules \( r_1, r_2, \ldots, r_m \) of the form

\[
r_k: \text{when } (\tau_k) \text{ may } s_k;
\]

What is \( \text{pre}(s, b) \)?

\[
\text{pre}(\text{while } (K) s, b) \equiv \text{formula } L_1 \lor L_2 \text{ for}
\]

\( L_1 \): those states that reach \( \neg K \land b \) with finite steps of \( s \)

through states in \( K \); and

\( L_2 \): those states that never leave \( K \) with steps of \( s \).
On-the-fly precondition calculation with GCM rules.

$L_1$: those states that reach $\neg K \land b$ with finite steps of $s$ through states in $K$

$w_0 = \neg K \land b$; $k = 1$;
repeat

$w_k = w_{k-1} \lor (K \land \text{pre}(s, w_{k-1}))$;
$k = k + 1$;
until $w_k \equiv w_{k-1}$;
return $w_k$.

Precondition to $b$ through while $(K) s$

Example: $b \equiv x \geq 2 \land y = 3$
while ($x < y$) $x = x + 1$;

$L_1$ computation.

$w_0 \equiv x \geq y \land x \geq 2 \land y = 3 \equiv \text{false}$; $k = 1$;

$w_1 \equiv \text{false} \lor (x < y \land \text{pre}(x = x + 1, \text{false}))$;
$\equiv \text{false} \lor (x < y \land \text{false})$;
$\equiv \text{false}$;
On-the-fly precondition calculation with GCM rules.

Given a set of rules $r_1, r_2, \ldots, r_m$ of the form

\[
\text{pre(while } (K) s, b) \]

$L_2$: those states that never leave $K$ with steps of $s$.

\[
\begin{align*}
  w_0 &= K; k = 1; \\
  \text{repeat} & \\
  w_k &= K \land \text{pre}(s, w_{k-1}); \\
  k &= k + 1; \\
  \text{until } w_k &= w_{k-1}; \\
  \text{return } w_k;
\end{align*}
\]

Precondition to $b$ through while $(K) s$;

Example:

while $(x < y \land x > 0)$ $x = x + 1$;

$L_2$ computation.

\[
\begin{align*}
  w_0 &= x < y \land x > 0; k = 1; \\
  w_1 &= x < y \land x > 0 \land \text{pre}(x = x + 1, x < y \land x > 0) \\
  &= x < y \land x > 0 \land x + 1 < y \land x + 1 > 0 \\
  \equiv x > 0 \land x + 1 < y
\end{align*}
\]

\[
\begin{align*}
  w_2 &= x + 1 < y \land x > 0 \land \text{pre}(x = x + 1, x + 1 < y \land x > 0) \\
  &= x + 1 < y \land x > 0 \land x + 2 < y \land x + 1 > 0 \\
  \equiv x > 0 \land x + 2 < y
\end{align*}
\]

non-terminating for algorithms and protocols!
Precondition to b through while (K) s;
Example:
while ( x>y && x>0)  x = x+1;
L2 computation.
\[ w_0 \equiv x>y \land x>0 ; \ k = 1; \]
\[ w_1 \equiv x>y \land x>0 \land \text{pre}(x=x+1, x>y \land x>0) \]
\[ \equiv x>y \land x>0 \land x+1>y \land x+1>0 \equiv x>y \land x>0 \]
terminating for algorithms and protocols!

Precondition to b through while (K) s;
Example: b \equiv x==2 \land y==3
while ( x>y && x>0)  x = x+1;
L1 computation.
\[ w_0 \equiv (x<=y \lor x<=0) \land x==2 \land y==3 \equiv x==2 \land y==3; \]
\[ w_1 \equiv (x==2 \land y==3) \lor (x>y \land x>0 \land \text{pre}(x=x+1, x==2 \land y==3)); \]
\[ \equiv (x==2 \land y==3) \lor (x>y \land x>0 \land x==1 \land y==3); \]
\[ \equiv (x==2 \land y==3) \lor \text{false} \]
\[ \equiv x==2 \land y==3 \]
Kripke structure - liveness analysis

Given
- a Kripke structure \( A = (S, S_0, R, L) \)
- a liveness predicate \( \eta \),

Can \( \eta \) be true eventually?

Example:
Can the computer be started successfully?
Will the alarm sound in case of fire?

Kripke structure - liveness analysis

Strongly connected component algorithm in graph theory

Given
- a Kripke structure \( A = (S, S_0, R, L) \)
- a liveness predicate \( \eta \),

find a cycle such that
- all states in the cycle are \( \neg \eta \)
- there is a \( \neg \eta \) path from a state in \( S_0 \) to the cycle.

Solutions in graph theory
- strongly connected components (SCC)
Kripke structure - liveness analysis

\[ (\text{Turn}=0 \rightarrow \Diamond \text{Turn}=1) \]

\[
\begin{align*}
\text{Turn}=0 & \quad L_0L_1 \\
& \rightarrow \quad L_0NC_1 \\
& \rightarrow \quad NC_0NC_1 \\
& \rightarrow \quad CR_0NC_1
\end{align*}
\]

\[
\begin{align*}
\text{Turn}=1 & \quad L_1L_1 \\
& \rightarrow \quad L_1NC_1 \\
& \rightarrow \quad NC_1CR_1 \\
& \rightarrow \quad NC_1NC_1
\end{align*}
\]

\[
\begin{align*}
\text{liveness}(\eta) & \quad /* \text{using greatest fixpoint algorithm} */ \{ \\
& \text{for all } s, \text{ if } \neg \eta \in L(s), \quad L(s)=L(s) \cup \{ \exists \Box \neg \eta \}; \\
& \text{repeat} \{ \\
& \quad \text{for all } s, \text{ if } \exists \Box \neg \eta \in L(s) \text{ and } \forall (s, s')(\exists \Box \neg \eta \not\in L(s)), \\
& \quad \quad \quad L(s)=L(s) - \{ \exists \Box \neg \eta \}; \\
& \text{} \} \text{ until no more changes to } L(s) \text{ for any } s. \\
& \text{if there is an } s_0 \in S_0 \text{ with } \exists \Box \neg \eta \in L(s_0), \\
& \quad \text{return ‘liveness not true,’} \\
& \quad \text{else return ‘liveness true.’} \\
& \} \\
\text{The procedure terminates since } S \text{ is finite in the Kripke structure.}
\]
Symbolic liveness analysis

Encode the states with variables $x_0, x_1, \ldots, x_n$.
- the state set as a proposition formula: $S(x_0, x_1, \ldots, x_n)$
- the non-liveness state set as $\neg \eta(x_0, x_1, \ldots, x_n)$
- the initial state set as $I(x_0, x_1, \ldots, x_n)$
- the transition set as $R(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n)$

\[
b_0 = \neg \eta(x_0, x_1, \ldots, x_n) \land S(x_0, x_1, \ldots, x_n); \quad k = 1;
\]

repeat
\[
b_k = b_{k-1} \land \exists x'_0 \exists x'_1 \ldots \exists x'_n (R(x_0, x_1, \ldots, x_n, x'_0, x'_1, \ldots, x'_n) \land b'_{k-1});
\]
\[
k = k + 1;
\]
until $b_k \equiv b_{k-1}$;

if $(b_k \land I(x_0, x_1, \ldots, x_n)) = \text{false}$, return ‘live’; else return ‘not live’;
Kripke structure - symbolic liveness analysis

states: \( S(x,y,z) \equiv (\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge y \wedge z) \equiv (\neg x) \vee (x \wedge \neg y) \)

initial state: \( I(x,y,z) \equiv \neg x \wedge \neg y \wedge \neg z \)

non-liveness state: \( \neg \eta(x,y,z) \equiv (\neg x) \vee (x \wedge \neg y \wedge z) \)

transitions: \( R(x,y,z,x',y',z') \equiv \)
\( (\neg x \wedge \neg y \wedge \neg z \wedge \neg x' \wedge \neg y' \wedge z') \vee (\neg x \wedge \neg y \wedge z \wedge \neg x' \wedge y' \wedge \neg z') \vee (\neg x \wedge y \wedge \neg z \wedge \neg x' \wedge y' \wedge \neg z') \vee (\neg x \wedge y \wedge z \wedge \neg x' \wedge y' \wedge \neg z') \vee (\neg x \wedge y \wedge z \wedge \neg x' \wedge y' \wedge z') \vee (\neg x \wedge y \wedge z \wedge x' \wedge \neg y' \wedge \neg z') \)
Symbolic liveness analysis

\[ b_0 = \neg \eta(x,y,z) \equiv (\neg x) \lor (x \land \neg y \land z) \]
\[ b_1 = b_0 \land \exists x' \exists y' \exists z'(R(x,y,z,x',y',z') \land b_0') \]
\[ = ((\neg x) \lor (x \land \neg y \land z)) \land \exists x' \exists y' \exists z'(R(x,y,z,x',y',z') \land ((\neg x') \lor (x' \land \neg y' \land z'))) \]
\[ = ((\neg x) \lor (x \land \neg y \land z)) \land \exists x' \exists y' \exists z'( ((\neg x \land \neg y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land \neg y \land z)) \land ((\neg x') \lor (x' \land \neg y' \land z')) \]
\[ = (\neg x \land \neg y \land z) \lor (\neg x \land y \land z) \lor (\neg x \land \neg y \land z) = \neg x \land (\neg y \lor \neg z) \]
\[ b_2 = b_1 \land \exists x' \exists y' \exists z'(R(x,y,z,x',y',z') \land b_1') \]
\[ = (\neg x \land \neg y \land z) \lor (\neg x \land y \land z) = \neg x \land \neg y \land z \]
\[ b_3 = b_2 \land \exists x' \exists y' \exists z'(R(x,y,z,x',y',z') \land b_2') \]
\[ = (\neg x \land \neg y \land \neg z) \lor (\neg x \land y \land \neg z) = \neg x \land \neg y \]

2009/12/16 stopped here.
Bisimulation Framework

**Design** implementation

- **Model construction**

- **Model Checker**

Answer **Yes** if the model is equivalent to the specification
**No** if not.

Bisimulation-checking

- $K = (S, S_0, R, L)$
- $K' = (S', S_0', R', L')$
- Note $K$ and $K'$ use the same set of atomic propositions $P$.
- $B \in S \times S'$ is a **bisimulation relation** between $K$ and $K'$ iff for every $B(s, s')$:
  - $L(s) = L'(s')$ (BSIM 1)
  - If $R(s, s_1)$, then there exists $s_1'$ such that $R'(s', s_1')$ and $B(s_1, s_1')$. (BISIM 2)
  - If $R(s', s_2')$, then there exists $s_2$ such that $R(s, s_2)$ and $B(s_2, s_2')$. (BISIM 3)
Bisimulations

Examples
Examples

Unwinding preserves bisimulation

Examples
Examples

![Graph Example 1]

Examples

![Graph Example 2]
Examples

\begin{center}
\begin{tikzpicture}
  \node[fill=orange] (p) {p} ;
  \node[fill=orange] (q1) [below of=p] {q} ;
  \node[fill=orange] (q2) [right of=q1] {q} ;
  \node[fill=orange] (r) [below of=q1] {r} ;
  \node[fill=orange] (s) [below of=q2] {s} ;
  \draw (p) -- (q1);
  \draw (q1) -- (r);
  \draw (q2) -- (s);
  \draw (p) -- (q2);
  \draw (q2) -- (s);
\end{tikzpicture}
\end{center}

186

Examples

\begin{center}
\begin{tikzpicture}
  \node[fill=orange] (p) {p} ;
  \node[fill=orange] (q1) [below of=p] {q} ;
  \node[fill=orange] (q2) [right of=q1] {q} ;
  \node[fill=orange] (r) [below of=q1] {r} ;
  \node[fill=orange] (s) [below of=q2] {s} ;
  \draw (p) -- (q1);
  \draw (q1) -- (r);
  \draw (q2) -- (s);
  \draw (p) -- (q2);
  \draw (q2) -- (s);
\end{tikzpicture}
\end{center}

187
Examples

Examples
Bisimulations

- $K = (S, S_0, R, L)$
- $K' = (S', S'_0, R', L')$
- $K$ and $K'$ are bisimilar (bisimulation equivalent) iff there exists a bisimulation relation $B \subseteq S \times S'$ between $K$ and $K'$ such that:
  - For each $s_0$ in $S_0$ there exists $s_0'$ in $S'_0$ such that $B(s_0, s_0').$
  - For each $s_0'$ in $S'_0$ there exists $s_0$ in $S_0$ such that $B(s_0, s_0').$

The Preservation Property.

- $K = (S, S_0, R, AP, L)$
- $K' = (S', S'_0, R', AP, L')$
- $B \subseteq S \times S'$, a bisimulation.
- Suppose $B(s, s').$

**FACT:** For any CTL* formula $\psi$ (over AP),

$K, s \vDash \psi$ iff $K', s' \vDash \psi.$

*If $K'$ is smaller than $K this is worth something.*

$\rightarrow$ abstraction for space reduction
Simulation Framework

Answer
Yes if the model satisfies the specification
No if not.

Simulation-checking

- \( K = (S, S_0, R, L) \)
  - \( K' = (S', S'_0, R', L') \)
- Note \( K \) and \( K' \) use the same set of atomic propositions \( AP \).
- \( B \subseteq S \times S' \) is a simulation relation between \( K \) and \( K' \) iff for every \( B(s, s') \):
  - \( L(s) = L'(s') \) (BSIM 1)
  - If \( R(s, s_1) \), then there exists \( s'_1 \) such that \( R'(s', s'_1) \) and \( B(s_1, s'_1) \). (BISIM 2)
Simulations

- \( K = (S, S_0, R, L) \)
- \( K' = (S', S'_0, R', L') \)
- \( K \) is simulated by (implements or refines) \( K' \) iff there exists a simulation relation \( B \subseteq S \times S' \) between \( K \) and \( K' \) such that for each \( s_0 \) in \( S_0 \) there exists \( s'_0 \) in \( S'_0 \) such that \( B(s_0, s'_0) \).

Bisimulation Quotients

- \( K = (S, S_0, R, L) \)
- There is a maximal simulation \( B \subseteq S \times S' \).
  - Let \( B \) be this bisimulation.
  - \( [s] = \{ s' \mid s B s' \} \).
- \( B \) can be computed “easily”.
- \( K' = K / B \) is the bisimulation quotient of \( K \).
**Bisimulation Quotient**

- \( K = (S, S_0, R, L) \)
- \([s] = \{s' \mid s \not B s'\}\).
- \(K' = K \div B = (S', S'_0, R', L')\).
  - \(S' = \{[s] \mid s \in S\}\)
  - \(S'_0 = \{[s_0] \mid s_0 \in S_0\}\)
  - \(R' = \{([s], [s']) \mid R(s_1, s_1'), s_1 \in [s], s_1' \in [s']\}\)
  - \(L'([s]) = L(s)\).

---

**Examples**

---

![Diagram](image.png)
Examples

![Diagram 1](image1.png)

Examples

![Diagram 2](image2.png)
Abstractions

- Bisimulations don’t produce often large reduction.
- Try notions such as simulations, data abstractions, symmetry reductions, partial order reductions etc.
- Not all properties may be preserved.
- They may not be preserved in a strong sense.

Graph Simulation

**Definition** Two edge-labeled graphs $G_1$, $G_2$

A *simulation* is a relation $R$ between nodes:
- if $(x_1, x_2) \in R$, and $(x_1,a,y_1) \in G_1$,
  then exists $(x_2,a,y_2) \in G_2$ (same label)
  s.t. $(y_1,y_2) \in R$

![Diagram of graph simulation]

Note: if we insist that $R$ be a function $\Rightarrow$ graph homeomorphism
Graph Bisimulation

**Definition** Two edge-labeled graphs $G_1$, $G_2$

A *bisimulation* is a relation $R$ between nodes s.t. both $R$ and $R^{-1}$ are simulations

---

Set Semantics for Semistructured Data

**Definition** Two rooted graphs $G_1$, $G_2$ are equal if there exists a bisimulation $R$ from $G_1$ to $G_2$ such that $(\text{root}(G_1), \text{root}(G_2)) \in R$

- Notation: $G_1 \approx G_2$

- For trees, this is precisely our earlier definition
Examples of Bisimilar Graphs

\[ \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} = \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} \]

Examples of non-Bisimilar Graphs

- This is a simulation but not a bisimulation
- Why?
- Notice: \( G_1, G_2 \) have the same sets of paths
Examples of Simulation

- Simulation acts like “subset”
  \[ \{a, b\} \subseteq \{a, b, c\} \]
  \[ \{a, b;\{c\}\} \subseteq \{d, a;\{e,f\}, b;\{c,g\}\} \]

- Question:
  - if \( DB_1 \subseteq DB_2 \) and \( DB_2 \subseteq DB_1 \) then \( DB_1 \approx DB_2 \) ?

Answer

If \( DB_1 \subseteq DB_2 \) and \( DB_2 \subseteq DB_1 \) then \( DB_1 \approx DB_2 \) ?

No. Here is a counter example:

\[ DB_1 \subseteq DB_2 \text{ and } DB_2 \subseteq DB_1 \text{ but NOT } DB_1 \approx DB_2 \]
Facts About a (Bi)Simulation

- The empty set is always a (bi)simulation
- If $R, R'$ are (bi)simulations, so is $R \cup R'$
- Hence, there always exists a maximal (bi)simulation:
  - Checking if $DB_1 = DB_2$: compute the maximal bisimulation $R$, then test $(\text{root}(DB_1), \text{root}(DB_2))$ in $R$

Computing a (Bi)Simulation

- Computing the maximal (bi)simulation:
  - $R = \{(s_1, s_2) | s_1 \in S_1, s_2 \in S_2, L_1(s_1) = L_2(s_2)\}$
  - While exists $(x_1, x_2) \in R$ that violates the definition, remove $(x_1, x_2)$ from $R$
- This runs in polynomial time! Better:
  - $O((m+n)\log(m+n))$ for bisimulation
  - $O(mn)$ for simulation
  - Compare to finding a graph homomorphism!
  - Compare to finding a graph isomorphism!

$\exists h((q,q') \in E \Rightarrow (h(q), h(q')) \in E')$
NP-Complete

$\exists h((q,q') \in E \iff (h(q), h(q')) \in E')$
NP-Hard?
Kripke structure - bisimulation analysis

bisimulation($K, K'$) /* using greatest fixpoint algorithm */ {
    $B = \{(s, s') | s \in S, s' \in S', L(s) = L(s')\}$;
    repeat {
        for all $(s, s') \in B$, {
            if $\exists (s, t) \in R, \forall (s', t') \in R'((t, t') \notin B)$, $B = B - \{(s, s')\}$;
            if $\exists (s', t') \in R', \forall (s, t) \in R((t, t') \notin B)$, $B = B - \{(s, s')\}$;
        }
    } until no more changes to $B$ for any $(s, s')$.
    if $\exists s_0 \in S_0 \forall s_0' \in S_0' ((s_0, s_0') \notin B)$, return "no bisimulation;"
    if $\exists s_0' \in S_0' \forall s_0 \in S_0 ((s_0, s_0') \notin B)$, return "no bisimulation;"
    return "bisimulation exists."
}

The procedure terminates since $B \subseteq S \times S'$ is finite.

A symbolic version is also possible. (skipped due to time-limit)

Language inclusion

Since both can be modeled as automata, we can check the relation between their languages.

- Language of a model: $L(\text{Model})$.
- Language of a specification: $L(\text{Spec})$.

We need: $L(\text{Model}) \subseteq L(\text{Spec})$. 
Language inclusion
- Correctness with runs

Language inclusion
- How to do it?

- Show that \( L(\text{Model}) \subseteq L(\text{Spec}) \).
- Equivalently:
  
  Show that \( L(\text{Model}) \cap L(\neg \text{Spec}) = \emptyset \).
- How? Check that \( A_{\text{model}} \cap A_{\neg \text{Spec}} \) is empty.

\[
\begin{array}{c}
\text{M} \quad \text{S} \\
\text{M} \subseteq \text{S} \\
\text{M} \cap \neg \text{S} \neq \emptyset
\end{array}
\]
Language inclusion
- What do we need to know?

L(Model) \cap L(\neg \text{Spec}) = \emptyset.

1. How to intersect two automata?
2. How to complement an automaton?
3. How to check for emptiness of an automaton?
4. How to translate from LTL to an automaton? (next week ...)

Language inclusion
- Automata for infinite sequences

State Sequences as Words
- Let $AP$ be the finite set of atomic propositions of the formula $f$.
- Let $\Sigma = 2^{AP}$ be the alphabet over $AP$.
- Every sequence of states is an $\omega$ word in $\Sigma^\omega$
  - $\alpha = P_0, P_1, P_2, \ldots$ where $P_i = L(s_i)$.
- A word $a$ is a model of formula $f$ iff $a \models f$
- Example: for $f = p \land (\neg q \cup q)$ \{p\},\{\},\{q\},\{p,q\}^\omega
- Let $\text{Mod}(f)$ denote the set of models of $f$. 
Language inclusion
- Büchi automata

Büchi automaton $A = (Q, \Sigma, \delta, I, F)$
- $Q$ – finite set of states
- $\Sigma$ – finite alphabet
- $\delta$ – transition relation
- $I$ – set of initial states
- $F$ – set of acceptance states

A run $\rho$ of $A$ on $\omega$ word $\alpha$

$\rho = q_0, q_1, q_2, \ldots$, s.t. $q_0 \in I$ and $(q_i, \alpha_i, q_{i+1}) \in \delta$

$\rho$ is accepting if $\text{Inf}(\rho) \cap F \neq \emptyset$

Buchi Automaton

- Given an infinite word $w \in \Sigma^\omega$ where $w = a_0, a_1, a_2, \ldots$
- a run $r$ of the automaton $A$ over $w$ is an infinite sequence of automaton states $r = q_0, q_1, q_2, \ldots$ where $q_0 \in I$ and for all $i \geq 0$, $(q_i, \alpha_i, q_{i+1}) \in \Delta$

- Given a run $r$, let $\text{inf}(r) \subseteq Q$ be the set of automata states that appear in $r$ infinitely many times

- A run $r$ is an accepting run if and only if $\text{inf}(r) \cap F \neq \emptyset$

  i.e., a run is an accepting run if some accepting states appear in $r$ infinitely many times
Transition System to Buchi Automaton Translation

Given a transition system $T = (S, I, R)$

- a set of atomic propositions $AP$
- a labeling function $L : S \times AP \rightarrow \{\text{true, false}\}$
the corresponding Buchi automaton $A_T = (\Sigma_T, Q_T, \delta_T, I_T, F_T)$

- $\Sigma_T = 2^{AP}$: an alphabet symbol corresponds to a set of atomic propositions
- $Q_T = S \cup \{i\}$: $i$ is a new state which is not in $S$
- $I_T = \{i\}$: $i$ is the only initial state
- $F_T = S \cup \{i\}$: all states of $A_T$ are accepting states

$\delta_T$ is defined as follows:

$$(s, a, s') \in \delta_T \text{ iff either } (s, s') \in R \text{ and } L(s', a) = \text{true} \text{ or } s=i \text{ and } s' \in I \text{ and } L(s', a) = \text{true}$$

Example transition system

Each state is labeled with the propositions that hold in that state
**Generalized Buchi Automaton**

A generalized Buchi automaton is a tuple $A = (\Sigma, Q, \delta, I, F)$ where

- $\Sigma$ is a finite alphabet
- $Q$ is a finite set of states
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation
- $I \subseteq Q$ is the set of initial states
- $F \subseteq 2^Q$ is sets of accepting states

i.e., $F = \{F_1, F_2, \ldots, F_k\}$ where $F_i \subseteq Q$ for $1 \leq i \leq k$

- Given a generalized Buchi automaton $A$, a run $r$ is an accepting run if and only if for all $1 \leq i \leq k$, $\text{inf}(r) \cap F_i \neq \emptyset$

**This is different than the standard definition**

---

**Buchi Automata Product**

Given $A_1=(\Sigma,Q_1,\delta_1,I_1,F_1)$ and $A_2=(\Sigma,Q_2,\delta_2,I_2,F_2)$

the product automaton $A_1 \times A_2 = (\Sigma, Q, \delta, I, F)$ is defined as:

- $Q = Q_1 \times Q_2$
- $I = I_1 \times I_2$
- $F = \{F_1 \times Q_2, Q_1 \times F_2\}$ (a generalized Buchi automaton)
- $\delta$ is defined as follows:
  - $((q_1,q_2),a,(q_1',q_2')) \in \delta$ iff $(q_1,a,q_1') \in \delta_1$ and $(q_2,a,q_2') \in \delta_2$

Based on the above construction, we get

$L(A_1 \times A_2) = L(A_1) \cap L(A_2)$
Example, a Special Case

Buchi automaton 1

Since all the states in the automaton 1 is accepting, only the accepting states of automaton 2 decide the accepting states of the product automaton

Buchi automaton 2

Product automaton

Buchi Automata Product Example

Automaton R

L(R) = (b^\omega a)^\omega

Automaton Q

L(Q) = (a^\omega b)^\omega

Automaton R \times Q

L(R \times Q) = L(R) \cap L(Q)

F = \{(r1,q1), (r1,q2),
      (r1,q1), (r2,q1)
    \}
Language inclusion - intersecting two finite-state automata

\[ A_1: \]

\[ A_2: \]

L(A_1) = (a+b) a + \varepsilon 
(words ending with 'a' + empty word)

L(A_2) = (ba)^* + (ba)^*b + \varepsilon 
(words that alternate between b and a + empty word)

What should be the language of \( A_1 \cap A_2 \)?

Language inclusion - intersecting two finite-state automata

A_1:

A_2:

1. States: \((s_0,t_0), (s_0,t_1), (s_1,t_0), (s_1,t_1)\).
2. Initial state(s): \((s_0,t_0)\).
3. Accepting states: \((s_0,t_0), (s_0,t_1)\).
Language inclusion
- intersecting two finite-state automata

\[ L(A_1 \cap A_2) = (ba)^* + \varepsilon \]

Language inclusion
- intersecting two Büchi automata

Previous method doesn’t work:
Language inclusion
- intersecting two Büchi automata

Strategy:
- “Multiply” the product automaton by 3
  \[(S = S_1 \times S_2 \times \{0,1,2\} )\]
- Start from the ‘0’ copy.
- Transition to the ‘1’ copy when visiting a state from F1
- Transition to the ‘2’ copy if in a ‘1’ state and visiting a state from F2, and in the next state back to a ‘0’ state.
- Make the ‘2’ copy an accepting set.

There are total of 12 states in the product automaton.
The reachable part of \(A_1 \cap A_2\) is:
Language inclusion
- How to complement?

- Complementation is hard!
- We know how to translate an LTL formula to a Buchi automaton. So we can:
  - Build an automaton $A$ for $\varphi$, and complement $A$, or
  - Negate the property, obtaining $\neg \varphi$ (the sequences that should never occur). Build an automaton for $\neg \varphi$.

Language inclusion
- How to check for emptiness?

- Need to check if there exists an accepting run (passes through an accepting state infinitely often).
- This is called checking for emptiness, because if no such run exists, then $L(A) = \epsilon$;
Language inclusion - emptiness and accepting runs

- If there is an accepting run, then it contains at least one accepting state an infinite number of times.
- This state must appear in a cycle.
- So, find a reachable accepting state on a cycle.
- What graph algorithm?

Language inclusion - Finding accepting runs

- Rather than looking for cycles, look for SCCs:
  - A Strongly Connected Component (SCC): a set of nodes where each node is reachable from all others.
  - Finding SCC’s is linear in the size of the graph.
  - Find a reachable SCC with an accepting node.
Language inclusion
- Verification under Fairness

Express the fairness as a property $\phi$. To prove a property $\psi$ under fairness, model check $\phi \rightarrow \psi$. 

Counterexample

Fair ($\phi$)

Bad ($\neg \psi$)

Program