Verification of Finite State Automata

- **Questions:**
  - Does the specification correctly describe the desired behavior?
  - Do specification and implementation match?
  - Can the system reach dangerous states?

- **Possible approaches:**
  - **Simulation** (validation): Success depends on right input patterns; can at most show the existence of some errors but not the absence.
  - **Formal Analysis** (verification): Formal proof of correctness.

Because of the finite number of states, verification is possible in principle by enumeration.

Because of the finite size of memory, the correctness of processors, software, communication systems, ... could be shown.

But is this a feasible approach?

<table>
<thead>
<tr>
<th># bits</th>
<th># states</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Bit</td>
<td>256</td>
</tr>
<tr>
<td>32 Bit</td>
<td>$4 \times 10^9$</td>
</tr>
<tr>
<td>1 kBit</td>
<td>$10^{300}$</td>
</tr>
<tr>
<td>1 Mbit</td>
<td>$10^{300,000}$</td>
</tr>
<tr>
<td>1 GBit</td>
<td>$10^{300,000,000}$</td>
</tr>
</tbody>
</table>

In recent years, there was a **break through** here!

**Symbolic Model Checking:**
- Formulation of questions in terms of logic formulas (temporal logic). **In this lecture, we will NOT cover this because of lack of time! Only a simple question will be tackled (reachability).**
- Efficient representation of sets and relations using OBDDs (ordered binary decision diagrams).

The methods are used in industry for proving the correctness of digital circuits (control path, arithmetic units) and of safety critical embedded systems (traffic control, airplane control, ...).
Principles

- **Comparison** of specification and implementation:
  - Reference
  - System under test
  - Data structure
  - Comparison

- **Proof of properties**:
  - System under test
  - Question
  - Data structure
  - Calculation (fixed point)
  - Answer

**Compare Specification and Implementation**

- **Problem 1**:
  - Specification using a Boolean function.
  - Implementation using a Boolean circuit.
  - Method (convert circuit into function, rewrite terms, normal forms ...)

- **Problem 2**:
  - Specification of a state machine using transition function.
  - Implementation using a Boolean circuit.
  - Method (unknown state encoding, huge # execution paths)

---

**Ordered Binary Decision Diagrams (OBDD)**

- OBDDs can be used to efficiently represent Boolean functions, sets, (output and transition) relations.

- Because of the unique representation of Boolean functions, they can be used to prove equivalence.

- Operations on Boolean functions can be done efficiently.

- They can be used only if sets, relations, ... are finite.

---

**Ordered Binary Decision Diagram (OBDD)**

- **Concept**:
  - Data structure for the representation of Boolean functions.
  - Unique (if reduced by removing redundant parts and if variable ordering is fixed).
  - Based on decision tree.

- **Form**:
  - Decision nodes that are associated to variables
  - Edges denote false (0, green) or true (1, red)
  - Leaves denote function values

\[
y = (x_1 + x_2) \cdot x_3
\]
Decomposition

- BDDs are based upon the Boole-Shannon-decomposition

\[ f = \overline{x} \cdot f|_{x=0} + x \cdot f|_{x=1} \]

- for each free variable, the function has two co-factors

\[ f|_{x=0} \text{ result for } x=0 \]
\[ f|_{x=1} \text{ result for } x=1 \]

Ordering of Variables

- Reduced BDDs are unique for a given fixed variable ordering.
- Therefore, ordered BDDs are used (OBDDs).
- The size of a BDD depends on the ordering (and can be exponential)

Calculations with BDDs

- **RESTRICT**: \( f|_{x=k} \)

  Operation: Delete edges corresponding to \( x = \overline{k} \) and apply simplification rules.

- **APPLY**: \( f < op > g \) with a Boolean operator \( op \)

  Operation: \( f \) and \( g \) are given as BDDs. Apply a recursive algorithm on \( f \) and \( g \) based on

\[ f < op > g = \overline{x} \cdot (f|_{x=0} < op > g|_{x=0}) + x \cdot (f|_{x=1} < op > g|_{x=1}) \]

Boolean expressions are converted to BDDs step by step.

\[ y = (x_1 \rightarrow x_2) \otimes x_3 \]

\[ y_1 = x_1 \rightarrow x_2 \]
\[ y = y_1 \otimes x_3 \]

Circuits are converted to Boolean functions first (based on a topological ordering of the gates).

Quantors are represented using APPLY and RESTRICT:

\[ \exists x : f(x) \leftrightarrow f(x)|_{x=0} + f(x)|_{x=1} = f(0) + f(1) \]
\[ \forall x : f(x) \leftrightarrow f(x)|_{x=0} \cdot f(x)|_{x=1} = f(0) \cdot f(1) \]
\[ \exists x_1, x_2 : f(x_1, x_2) \leftrightarrow \exists x_1 : (\exists x_2 : f(x_1, x_2)) \]
\[ \forall x_1, x_2 : f(x_1, x_2) \leftrightarrow \forall x_1 : (\forall x_2 : f(x_1, x_2)) \]
Sets and Relations

- **Representation of a subset** \( A \subseteq E \):
  - Binary coding \( \sigma(e) \) of elements \( e \in E \)
  - \( A \) is represented by characteristic function \( \psi_A(\sigma(a)) \)

- **Operations on sets**:
  - \( c \in A \cap B \leftrightarrow \psi_A(\sigma(c)) \cdot \psi_B(\sigma(c)) \)
  - \( c \in A \cup B \leftrightarrow \psi_A(\sigma(c)) + \psi_B(\sigma(c)) \)
  - \( c \in A - B \leftrightarrow \psi_A(\sigma(c)) \cdot \overline{\psi_B(\sigma(c))} \)

- **Example**:
  \( \psi_A = x_0 \otimes x_1 \leftrightarrow A = \{01',10'\} \)

Sets and Relations

- **Representation of a relation** \( R \subseteq A \times B \):
  - Binary coding \( \sigma(a), \sigma(b) \) of elements \( a \in A, b \in B \)
  - \( R \) is represented by

\[
\begin{align*}
\psi_R(\sigma(a), \sigma(b)) &= 0 \\
\psi_R(\sigma(a), \sigma(b)) &= 1
\end{align*}
\]

- **Example finite state automaton**:

\[
\begin{align*}
\psi_f(u, x, x') &= 1 \\
\psi_g(u, x, y) &= 1
\end{align*}
\]

Equivalence of Boolean Circuits

- **Comparison** between specification and implementation or between two implementations.

- **Method**:
  - Represent the two systems as OBDDs by applying the APPLY operator repetitively.
  - Compare structure of OBDDs.

- **Example**:
  \( y = (x_1 + x_2) \cdot x_3 \)

Reachable States

- **Problem**: Is a state \( x \in X \) reachable?

- **Solution**:
  - Represent state sets and transition relations as OBDDs.
  - Transform sets of states.
  - Iterative transition until a stable set of states is obtained.

\[
\begin{align*}
X_0 &= \{x_0\} \\
X_1 &= X_0 \cup \{x_1\} \\
X_2 &= X_1 \cup \{x_1, x_2\} \\
X_3 &= X_2 \cup \{x_1, x_2\}
\end{align*}
\]
Reachable States

- **Core transformation:**
  - Determine the set of all direct successor states of a given state set \( X \) using transition relation \( f \):
  \[
  X' = \text{Im}(X, f) = \{ x' : \exists x \text{ mit } \psi_X(x) \land \psi_f(x, x') \}
  \]

- **Calculation:**
  \[
  h(x, x') = \psi_X(x) \cdot \psi_f(x, x')
  \]
  \[
  \psi_{X'}(x') = (\exists x : h(x, x'))
  \]
  Calculation using OBDDs

Reachable States

- **Fixed point calculation:**
  - Starting from a set of initial states, determine the set of states that can be reached in \textbf{one or several steps}:
  \[
  X_0 = \{ x_0 \}
  \]
  \[
  X_{i+1} = X_i \cup \text{Im}(X_i, f) \text{ until } X_{i+1} = X_i
  \]
  \[
  \psi_{X_{i+1}}(x') = \psi_{X_i}(x') + (\exists x : \psi_{X_i}(x) \cdot \psi_f(x, x'))
  \]
  - Because of the finite set of states, a fixed point exists and is reached in finite time.
  - Test whether a state is reachable using resulting BDD.

Equivalence of Finite State Automata

- A method \textbf{based on reachability} is described:

  - Calculate the reachable states of the combined automaton.
  - Compare the outputs for equality.

- Calculate the common transition function:
  \[
  \psi_f(x_1, x_2, x_1', x_2') = (\exists u : \psi_{f_1}(u, x_1, x_1') \cdot \psi_{f_2}(u, x_2, x_2'))
  \]

- Determine the set of reachable states (as before):
  \[
  \psi_X(x_1, x_2)
  \]

- Determine the set of reachable output values:
  \[
  \psi_Y(y_1, y_2) = (\exists x_1, x_2 : \psi_X(x_1, x_2) \cdot \psi_{g_1}(x_1, y_1) \cdot \psi_{g_2}(x_2, y_2))
  \]

- Automata are different if the following term is true:
  \[
  \exists y_1, y_2 : \psi_Y(y_1, y_2) \cdot (y_1 \neq y_2)
  \]
Verification of Finite State Automata

- **Check time properties** of a finite state automaton, for example:
  1. Can a reset state reached from every reachable state?
  2. Is every request followed by an acknowledgement, eventually?
  3. Are the outputs equal for all reachable states?

- Usually, these questions are formulated by an expression in some **temporal logic**, for example CTL (computation tree logic).

- **Operators and quantors**:
  - $X$: in the next step; $F$: eventually; $G$: every times
  - $A$: for all paths; $E$: for at least one path

We will not explore this further ….

Concluding Remarks

- **Possible extensions**:
  - Proof of properties in absolute time using the concept of clocks.
  - Verification of systems with a potentially unlimited number of states.
  - Combination of discrete event systems and systems with continuous state (hybrid systems).

- Public domain software available, e.g. **SMV**:
  - General input language for system specification.
  - Accepts CTL formulas.
  - Produces counter examples.

Example: Counter Verification with SMV

```plaintext
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
SPEC AF bit2.carry_out
  -- "For all execution paths, the value of bit2.carry_out will eventually be false." This will be true.
SPEC AG !bit2.carry_out
  -- "For all execution paths, the value of bit2.carry_out will be false every times."
  -- This will be false and a counter example will be produced.

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := (value + carry_in) mod 2;
DEFINE
  carry_out := value & carry_in;
```