Deterministic Finite Automata

COMP2600 — Formal Methods for Software Engineering

Zhē Hóu

Australian National University
Semester 2, 2014 (Slides created by Ranald Clouston)
You can call me

“Zee”, or

“Zeee”, “Zeeeee”,…

or (a bit more) formally, \( Zee^* \), where we use \( * \) as a superscript to mean “occurring 0 or more times”

or \( Zee^+ \), where we use \( + \) as a superscript to mean “occurring 1 or more times”
Pop quiz

What is

\[ x - y \left( \frac{x}{y} \right) \]

where

- \( x = 4195835 \);
- \( y = 3145727 \)?
Pop quiz - answer

If you answered 256, congratulations! You are a Pentium microprocessor released by Intel in 1993.

This floating point error evaded Intel’s testing, only to be discovered by computational number theorist Thomas R. Nicely while generating large prime numbers.

A full recall of the chip ended up costing Intel US$475 million.

But how can Intel ever have confidence that their hardware is error-free and so won’t cost them money and reputation?

Sounds like a job for formal methods!
Formal modelling

Intel are now world leaders in the formal verification of hardware.

A full description of the formal methods approaches they use is outside the bounds of this course.

But here’s a specific problem - how do you reason **mathematically** about a **physical** object like a computer chip?

The solution is to model the physical object via a mathematically defined **abstract machine**.
The big picture

<table>
<thead>
<tr>
<th>Languages</th>
<th>Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Language/</td>
<td>Deterministic Finite Automata/</td>
</tr>
<tr>
<td>Regular Expression</td>
<td>Non-deterministic Finite Automata</td>
</tr>
<tr>
<td>Context-free Language</td>
<td>Pushdown Automata</td>
</tr>
<tr>
<td>Recursively Enumerable Language</td>
<td>Turing Machine</td>
</tr>
</tbody>
</table>
Machine vs. Language

Abstract machines in this course will

- Accept as \textbf{input} a \textit{string} of symbols.
- Produce as \textbf{output} \textit{success} or \textit{failure} only.

Therefore an abstract machine determines a set of strings it accepts.

Conversely, given a set of strings (called a \textit{language}) we might try to design an abstract machine that recognises all and only those strings.

The study of abstract machines and the study of (formal) languages are therefore closely linked:

- Given a machine, find the language that is accepted by this machine.
- Given a language, find the machine that accepts this language.
Other correspondences

Machines vs. Languages
How about, for example, DFA vs. Regular Language?

- For any deterministic finite automaton, there is a regular language that is accepted by it.
- For any regular language, there is a deterministic finite automaton that accepts it.

Language vs. Problem
In the formal language field, languages and problems are the same thing.

- A language $L$ is a set of strings.
- A problem $P_L$ of language $L$ is to ask, given a string $w$, is it in $L$?
Language Examples

Formal Language Theory gets applied to natural and artificial languages and to programming languages, but in this course we will look at simpler examples over small alphabets, e.g.:

1. finite sets.
   
   \{a, aa, ab, aaa, aab, aba, abb\}

2. Palindromes consisting of bits (0,1):
   
   \{0, 1, 00, 11, 010, 101, 000, 111, 0110, ...\}

In each case the language is the set of strings. The first is a finite language and the second is an infinite language.
Language Terminology

- The **alphabet** (or **vocabulary**) of a formal language is a set of **tokens** (or **letters**). It is usually denoted $\Sigma$.

- A **string** (or **word**) over $\Sigma$ is a **sequence** of tokens, or the null-string $\varepsilon$.
  
  For example, if $\Sigma = \{a, b, c\}$, then $ababc$ is a string over $\Sigma$.

- A **language** with alphabet $\Sigma$ is some set of strings over $\Sigma$.

**Notation:**

- $\Sigma^*$ is the set of all strings over $\Sigma$.

- Therefore, every language with alphabet $\Sigma$ is some **subset** of $\Sigma^*$.
Finite State Automata

The first flavour of abstract machine we will look at will be the least powerful - Finite State Automata (FSA), i.e.,

- Deterministic Finite Automata (DFA), or
- Non-deterministic Finite Automata (NFA)

They are still powerful enough to encode the actions of microprocessors, and so are a fundamental formal methods concept applied by Intel and others.

In fact they were invented, not by computer scientists interested in hardware, but by two neuropsychologists, Warren S. McCullough and Walter Pitts, who back in 1943 were trying to model neurons in the human brain!

Such unlikely connections are a common theme in the history of science.
FSA (and Regular Language) - other applications

- Traffic lights
- Vending machines
- Games
- Simple pattern matching, such as checking the validity of
  - email addresses
  - user names
  - passwords
Informal definition of Finite State Automata

The simplest useful abstraction of a “computing machine” consists of:

- A fixed, finite set of states
- A transition relation over the states

Simplified Example: a traffic light machine has 3 states:

Some information is missing – where does the machine start and stop, and what triggers transitions between states?
Example: Vending Machine

Imagine a vending machine which (1) accepts $1 and $2 coins; (2) refunds all money if more than $4 is added; (3) is ready to deliver if exactly $4 has been added.

Note that the transitions are labelled with important information, and the start state and end state(s) are specially labelled.
The Vending Machine ctd

- You start at the state “$0”, indicated by the $\rightarrow$ at the left
- the alphabet $\Sigma = \{1, 2\}$
- at the “$4$” state (circled) you have credit for a purchase
- what strings of tokens (starting at the “$0$”) leave you in the circled state?
  - the empty string $\varepsilon$ $\times$
  - $22$ $\checkmark$
  - $1222$ $\times$
  - $1222221111$ $\checkmark$
- the accepted strings are the language defined by the automaton.
Terminology

We will study deterministic finite automata (DFA) first.

- The alphabet of a DFA is a finite set of input tokens that an automaton acts on.
- a DFA has a finite set of states.
- One of the states is the initial state — where the automaton starts.
- At least one of the states is a final state.
- A transition function (next state function):

\[ \text{State} \times \text{Token} \rightarrow \text{State} \]
**Formal Definition of DFA**

A Deterministic Finite State Automaton (DFA) is completely characterised by five items:

\[(\Sigma, S, s_0, F, N)\]

- an input alphabet \(\Sigma\), the set of tokens
- a finite set of states \(S\)
- an “initial” state \(s_0 \in S\) (we start here)
- a set of “final” states \(F \subseteq S\) (we hope to finish in one of these)
- a transition function \(N : S \times \Sigma \rightarrow S\)

(We will see later that \(N\) being a function is the reason the automaton is deterministic.)
Example 1

- Alphabet - \{0, 1\}
- States - \{S_0, S_1, S_2\}
- Initial state - $S_0$
- Final states - \{S_2\}
- Transition function (as a table) -

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
</tbody>
</table>

(Note that the actual state names are irrelevant.)
Example 1, ctd

If $N$ is the transition function, then we write $N(S_0, 0)$ when we want to refer to the state that the above automaton goes to when it starts in $S_0$ and absorbs a $0$.

From the definition, $N(S_0, 0) = S_1$.

Similarly, $N(S_1, 1) = S_2$.
Eventual State Function

Revisit example 1:

- Input 0101 takes the DFA from $S_0$ to $S_2$,
  Input 1011 takes the DFA from $S_1$ to $S_0$, etc.

- A complete list of such possibilities is a function from a given state and a string to an ‘eventual state.’

This is the automaton’s Eventual State Function.
Eventual State Function — Definition

Suppose $A$ is a DFA with states $S$, alphabet $\Sigma$, and transition function $N$. The eventual state function for $A$ is

$$N^* : S \times \Sigma^* \rightarrow S$$

$N^*(s, w)$ is the state $A$ reaches, starting in state $s$ and reading string $w$.

$N^*$ can be defined inductively:

1. $N^*(s, \varepsilon) = s$ \hspace{1cm} (N1)
2. $N^*(s, x\alpha) = N^*(N(s, x), \alpha)$ \hspace{1cm} (N2)

It is the eventual state function acting on the start state that we are really interested in.
An Important (but Unsurprising) Theorem about $N^*$

The “Append” Theorem

For all states $s \in S$ and for all strings $\alpha, \beta \in \Sigma^*$

$$N^*(s, \alpha\beta) = N^*(N^*(s, \alpha), \beta)$$

**Proof** by induction on the length of $\alpha$.

Base case: $\alpha = \varepsilon$

$$\text{LHS} = N^*(s, \varepsilon\beta) = N^*(s, \beta)$$

$$\text{RHS} = N^*(N^*(s, \varepsilon), \beta)$$

$$= N^*(s, \beta) = \text{LHS}$$  \hspace{1cm} (by (N1))
Proof ctd: Step case:

Suppose \( N^*(s, \alpha\beta) = N^*(N^*(s, \alpha), \beta) \)  

\[ \text{LHS} = N^*(s, (x\alpha)\beta) \]
\[ = N^*(s, x(\alpha\beta)) \]
\[ = N^*(N(s, x), \alpha\beta) \]  \hspace{1cm} (by (N2))
\[ = N^*(N^*(N(s, x), \alpha), \beta) \]  \hspace{1cm} (by IH)

\[ \text{RHS} = N^*(N^*(s, x\alpha), \beta) \]
\[ = N^*(N^*(N(s, x), \alpha), \beta) \]  \hspace{1cm} (by (N2))

Corollary — when \( \beta \) is a single token

\[ N^*(s, \alpha y) = N(N^*(s, \alpha), y) \]
Example

\[ N^*(S_1, 1011) = N^*(N(S_1, 1), 011) \]
\[ = N^*(S_2, 011) \]
\[ = N^*(S_1, 11) \]
\[ = N^*(S_2, 1) \]
\[ = N^*(S_0, \varepsilon) \]
\[ = S_0 \]