Deterministic Finite Automata

COMP2600 — Formal Methods for Software Engineering

Ranald Clouston
Australian National University
Semester 2, 2013

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\$175 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?
Abstract Machines

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

This approach is also useful for software, and is particularly well suited to certain applications such as parsing.

Abstract machines come in various flavours, and will be the focus of the rest of this course.

The technical highlight will be Turing machines, which are powerful enough to perform any computation that can be imagined!

However less powerful abstract machines are often easier to reason about and hence more useful for formal methods applications.

Finite State Automata

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

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.

Input and Output

Finite state automata in this course will

- Accept as input a string of symbols.
- Produce as output success or failure only.

Therefore one way to think about abstract machines is to ask which strings are accepted and which rejected by them.

Conversely, given a set of strings (called a 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.

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 $\Sigma$.

• The strings of a language are called the **sentences** of the language.

**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 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:

- **G** names state in which light is green.
- **Y** names state in which light is yellow.
- **R** names state in which light is red.

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.

![Vending Machine Diagram]

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 ✓
  - 1222 ×
  - 122221111 ✓

• 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:

\[ (\Sigma, S, s_0, F, \mathcal{N}) \]

- 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 \( \mathcal{N} : S \times \Sigma \rightarrow S \)

(We will see later that \( \mathcal{N} \) 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) -

\[
\begin{array}{c|cc}
   & 0 & 1 \\
\hline
S_0 & S_0 & S_1 \\
S_1 & S_1 & S_2 \\
S_2 & S_1 & S_0 \\
\end{array}
\]

(Note that the actual state names are irrelevant.)

Example 1, ctd

If \( \mathcal{N} \) is the transition function, then we write \( \mathcal{N}(S_0, 0) \) 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, \( \mathcal{N}(S_0, 0) = S_1 \).

Similarly, \( \mathcal{N}(S_1, 1) = S_2 \).
Eventual State Function

Revisit example 1:

- \( S_0 \)
- \( S_1 \)
- \( S_2 \)

- 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:

\[
\begin{align*}
N^* (s, \varepsilon) &= s \\
N^* (s, x\alpha) &= N^* (N^* (s, x), \alpha)
\end{align*}
\]

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: \( \alpha = \varepsilon \)

\[
\begin{align*}
\text{LHS} &= N^* (s, e\beta) = N^* (s, \beta) \\
\text{RHS} &= N^* (N^* (s, \varepsilon), \beta) \\
&= N^* (s, \beta) \quad \text{(by (N1))}
\end{align*}
\]

Proof ctd: Step case:

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

\[
\begin{align*}
\text{LHS} &= N^* (s, (x\alpha)\beta) \\
&= N^* (N^* (s, x\alpha), \beta) \\
&= N^* (N^* (N^* (s, x), \alpha), \beta) \quad \text{(by (N2))}
\end{align*}
\]

\[
\begin{align*}
\text{RHS} &= N^* (N^* (s, x\alpha), \beta) \\
&= N^* (N^* (N^* (s, x), \alpha), \beta) \quad \text{(by (N2))}
\end{align*}
\]

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

\[
N^* (s, \alpha y) = N(N^* (s, \alpha), y)
\]
Language of an Automaton

We say a DFA accepts a string if, starting from the start state, it terminates in one of the final states. More precisely, let $A = (\Sigma, S, s_0, F, N)$ be a DFA and $w$ be a string in $\Sigma^*$. We say $w$ is accepted by $A$ if

$$N^*(s_0, w) \in F$$

The language accepted by $A$ is the set of all strings accepted by $A$:

$$L(A) = \{ w \in \Sigma^* | N^*(s_0, w) \in F \}$$

That is, $w \in L(A)$ iff $N^*(s_0, w) \in F$.

Example 1 again

It is easy to see whether a given string is accepted or not.

For example, 0011101 takes the machine from state $S_0$ through states $S_1$, $S_2$, $S_0$, $S_0$, $S_1$ to $S_2$ (a final state).

$N^*(S_0, 0011101) = N^*(N(S_1, 1), 011) = N^*(S_2, 011) = N^*(S_1, 11) = N^*(S_2, 1) = N^*(S_0, \epsilon) = S_0$

Other strings that are accepted: 01, 001, 101, 0001, 0101, 00101101 . . .

Not accepted: $\epsilon, 0, 1, 00, 10, 11, 100, \ldots$

How is the first list different from the second list...?

How do we justify our guess at this answer?
Proving an Acceptance Predicate — in General

If we are to claim that a machine $M$ accepts the language that is characterised by a predicate $P$, then we must prove the following:

**Proof obligation 1:**
Show that any string satisfying $P$ is accepted by $M$.

**Proof obligation 2:**
Show any string accepted by $M$ satisfies $P$.

---

**Part 1:** \( \forall \alpha \in \Sigma^*, N^*(S_0, \alpha 01) \in F \)

**Lemma:**
\[ \forall s \in S, N^*(s, 01) = S_2 \]

**Proof:** Prove that \( N^*(s, 01) = S_2 \) by cases:
\[ N^*(S_0, 01) = N^*(S_1, 1) = S_2 \]
\[ N^*(S_1, 01) = N^*(S_1, 1) = S_2 \]
\[ N^*(S_2, 01) = N^*(S_1, 1) = S_2 \]

So, by the “append” theorem above, whether \( N^*(S_0, \alpha) \) is \( S_0, S_1 \) or \( S_2 \),
\[ N^*(S_0, \alpha 01) = N^*(N^*(S_0, \alpha), 01) = S_2 \]

**Part 2:** \( N^*(S_0, w) = S_2 \implies \exists \alpha. w = \alpha 01 \)

**Proof:**
- If \( w \) were the empty string \( \varepsilon \), then \( N^*(S_0, \varepsilon) = S_0 \neq S_2 \), so the antecedent must be false, so by the rules of propositional logic, the claim is true!
- If \( w \) were one token long then again the antecedent is false - there is no one step path from \( S_0 \) to \( S_2 \).
- There clearly are paths from \( S_0 \) to \( S_2 \) of two or more steps, so suppose we are in that position with \( N^*(S_0, \alpha \varepsilon y) = S_2 \).

Then by the corollary to the “append” theorem,
\[ N(N^*(S_0, \alpha), y) = S_2 \]

By the definition of \( N \), \( y \) must be \( 1 \) and \( N^*(S_0, \alpha x) \) must be \( S_1 \).
Similarly, \( N(N^*(S_0, \alpha), x) = S_1 \), so \( x \) is \( 0 \), again by the definition of \( N \).
Another Example

What language does this DFA accept?

Answer for $O1B$

$O1B$ accepts the language of bit-strings containing exactly one 1-bit.

Proof obligations:

- Show that if a bit-string contains exactly one 1-bit then it is accepted by $O1B$.
- Show that if a string is accepted by $O1B$ it contains exactly one 1-bit.

Mapping to Mathematics

Expressed mathematically, the main conclusion is

$$L(O1B) = \{ w \in \Sigma^* \mid w = 0^m10^n \}$$

where $m, n$ are any natural numbers.

The two subgoals are

1. If $w = 0^n10^n$ then $N^*(S_0, w) = S_1$
2. If $N^*(S_0, w) = S_1$ then $w = 0^n10^n$.

For this DFA the phrase "$w$ is accepted by $O1B$" is captured by the expression $N^*(S_0, w) = S_1$.

Proving these subgoals

The first subgoal follows immediately from the following two lemmas, which are easily proved by induction:

$$\forall n \geq 0. N^*(S_0, 0^n) = S_0$$
$$\forall n \geq 0. N^*(S_1, 0^n) = S_1$$

Therefore

$$N^*(S_0, 0^n10^m) = N^*(N^*(S_0, 0^n), 10^m) = N^*(S_0, 10^m)$$
$$= N^*(N(S_0, 1), 0^m) = N^*(S_1, 0^m) = S_1$$

The second subgoal, stated more formally as

$$\forall w : N^*(S_0, w) = S_1 \implies \exists n, m \geq 0. w = 0^n10^m$$

is proved on the next slide.
Proof by contradiction (ctd)

Suppose \( A \) is a DFA that accepts \( L \). That is \( L = L(A) \).

Each of the following expressions denotes a state of \( A \)
\[
N^*(S_0, a), N^*(S_0, aa), N^*(S_0, a^3) \ldots
\]

Since this list is infinite and the number of states in \( A \) is finite, some of these expressions must denote the same state.

Choose distinct \( i \) and \( j \) such that \( N^*(S_0, a^i) = N^*(S_0, a^j) \).

(What we have done here is pick on two initial string fragments that the automaton will not be able to distinguish, in terms of what is allowed for the rest of the string)

Limitations of DFAs

General Question: How powerful are DFAs?
Specifically, what class of languages can be recognised by DFAs?

A very important example:

Consider this language: \( L = \{ a^n b^n \mid n \in \mathbb{N} \} \)
That is, \( L = \{ \varepsilon, ab, aabb, aabbbb, a^4 b^4, a^5 b^5, \ldots \} \)
This language cannot be recognised by any finite state automata
This is because the memory of a DFA is limited, but the machine would have to remember how many ‘a’s it has seen.

Proof by contradiction

Suppose \( A \) is a DFA that accepts \( L \). That is \( L = L(A) \).

Each of the following expressions denotes a state of \( A \)
\[
N^*(S_0, a), N^*(S_0, aa), N^*(S_0, a^3) \ldots
\]

Since this list is infinite and the number of states in \( A \) is finite, some of these expressions must denote the same state.

Choose distinct \( i \) and \( j \) such that \( N^*(S_0, a^i) = N^*(S_0, a^j) \).

(What we have done here is pick on two initial string fragments that the automaton will not be able to distinguish, in terms of what is allowed for the rest of the string)
Pigeon-Hole Principle

The proof used the pigeon-hole principle:

If we have more pigeons than pigeon-holes, then at least two pigeons must be sharing a hole.

This was obviously true in our example as we had infinitely many pigeons crammed into finitely many holes.

This is a useful generic way to prove that no finite state automata can recognise certain infinite languages.

(but there are many infinite languages that can be recognised by DFAs, as we’ve seen, so be careful!)

Equivalence of Automata

Two automata are said to be equivalent if they accept the same language.

Interesting question:
Can we simplify a DFA? Is there an equivalent DFA with fewer states?

Equivalence of States

Two states $S_j$ and $S_k$ of an DFA are equivalent if, for all input strings $w$

$$N^*(S_j, w) \in F \text{ if and only if } N^*(S_k, w) \in F$$

Note that $N^*(S_j, w)$ and $N^*(S_k, w)$ may be different states - we only care that both are in, or not in, $F$.

In the following example, $S_2$ is equivalent to $S_4$.

An Algorithm for Finding Equivalent States

There is an iterative algorithm to compute a list of equivalence classes of states.

- The working data structure for the algorithm is a list of lists (“groups”) of states
- Each group contains states that appear to be equivalent, given the tests we have done so far
- On each iteration, we test one of the groups with a symbol from the alphabet.
- If we notice differing behaviour, we split the group.
The Algorithm Details

- **Input**: A list containing two “groups” (a group is represented as a list of states). One group consists of the Final states and the other consists of the non-final states.
- **Data**: The working data structure, \( WDS : [ [ \text{State} ] ] \), is a list of groups of states. When two states are in different groups, we know they are not equivalent.
- **Loop**: Pick a group, \( \{ s_1, ..., s_j \} \) and a symbol, \( x \). If the states \( N( s_i, x ) \) are all in the same group, then the group \( \{ s_1, ..., s_j \} \) is not split. If the states \( \{ N( s_i, x ) \} \) belong to different groups of \( WDS \), then the group \( \{ s_1, ..., s_j \} \) should be split accordingly.
- **Continue until** we cannot, by *any* choice of letter, split *any* group.

Elimination of States

Suppose \( A = ( \Sigma, S, S_0, F, N ) \) is a DFA with state \( S_k \) equivalent to \( S_j \).

We can eliminate \( S_k \) from this automaton by defining a new automaton \( A' = ( \Sigma, S', S_0, F', N' ) \) as follows:

- \( S' \) is \( S \) without \( S_k \)
- \( F' \) is \( F \) without \( S_k \)
- \( N'(s, w) = \begin{cases} S_j & \text{if } N(s, w) = S_k \\ N(s, w) & \text{otherwise.} \end{cases} \)

Our Previous Example

We split the non-final states, but leave the final states together:

\[
\begin{align*}
A_2: & \\
S_0 & \xrightarrow{a} S_1 \\
S_0 & \xrightarrow{b} S_3 \\
S_1 & \xrightarrow{a} S_2 \\
S_1 & \xrightarrow{b} S_4 \\
S_2 & \xrightarrow{a} [s_0, s_1, s_3] \\
S_2 & \xrightarrow{b} [s_0, s_2, s_4] \\
S_3 & \xrightarrow{a} [s_0, s_1, s_3, s_2, s_4] \\
S_3 & \xrightarrow{b} [s_0, s_1, s_3, s_2, s_4] \\
S_4 & \xrightarrow{a} [s_0, s_1, s_3, s_2, s_4] \\
S_4 & \xrightarrow{b} [s_0, s_1, s_3, s_2, s_4] \\
S_0 & \xrightarrow{a} [s_0, s_1, s_3] \\
S_0 & \xrightarrow{b} [s_0, s_2, s_4] \\
S_1 & \xrightarrow{a} [s_0, s_1, s_3] \\
S_1 & \xrightarrow{b} [s_0, s_2, s_4] \\
S_2 & \xrightarrow{a} [s_0, s_1, s_3] \\
S_2 & \xrightarrow{b} [s_0, s_2, s_4] \\
S_3 & \xrightarrow{a} [s_0, s_1, s_3] \\
S_3 & \xrightarrow{b} [s_0, s_2, s_4] \\
S_4 & \xrightarrow{a} [s_0, s_1, s_3] \\
S_4 & \xrightarrow{b} [s_0, s_2, s_4]
\end{align*}
\]

Example

Since \( S_2 \equiv S_4 \) in \( A_2 \), let’s eliminate \( S_4 \).

- New set of states is \( \{ S_0, S_1, S_2, S_3 \} \)
- New set of final states is \( \{ S_2 \} \)
- New transition function is:

\[
A_3: \begin{align*}
S_0 & \xrightarrow{a} S_1 \\
S_0 & \xrightarrow{b} S_3 \\
S_1 & \xrightarrow{a} S_2 \\
S_1 & \xrightarrow{b} a, b \\
S_2 & \xrightarrow{a} S_1 \\
S_2 & \xrightarrow{b} a, b \\
S_3 & \xrightarrow{a} S_1 \\
S_3 & \xrightarrow{b} a, b \\
S_1 & \xrightarrow{a} S_2 \\
S_1 & \xrightarrow{b} a, b \\
S_2 & \xrightarrow{a} S_1 \\
S_2 & \xrightarrow{b} a, b \\
S_3 & \xrightarrow{a} S_1 \\
S_3 & \xrightarrow{b} a, b
\end{align*}
\]
**DFA Minimisation**

Consider the DFA below:

None of $S_1$, $S_2$, $S_3$ are equivalent...

But $S_3$ is *inaccessible* from the start state so can safely be deleted (along with the transitions emerging from it).

Deleting equivalent and inaccessible states will give a *minimal* DFA.