Formal Methods: lecture 2 notes
written down by Przemysław Daca

October 9, 2012

1 Automata

Automata, or state transition systems, are a method of defining languages that is alternative to formal systems. We will introduce this concept on an example.

Imagine that on one side of a river there is a wolf, goat, cabbage and a boat. The goal is to transport them all to the other side on condition that the boat can carry only one thing at a time. The captain of the boat keeps an eye on the animals and makes sure that nothing gets eaten. As a consequence it is not allowed to leave the wolf and the goat or the goat and the cabbage on the same side without the boat.

Figure 1: Illustration from [http://www.cs.ru.nl/~fvaan/](http://www.cs.ru.nl/~fvaan/)

To solve this problem we formalize it as an automaton that is shown as a graph in Figure 2. Nodes of the graph are called states and tell us where the actors are. The text above and below the vertical line represent sides of the river. The top state is pointed by a small arrow, which means it is the initial one. The arrows between states are called transitions and they represent transporting items across the river. Transitions $g$, $w$ and $c$ express that the corresponding item was transported and “−” means that nothing was carried. The target state, where all
actors are on the other side, is at the bottom of the figure. It is called an accepting state and is indicated by a double circle. Some transitions result immediately in losing the game, for instance transporting the cabbage from the initial state will cause the goat to be eaten. Some, but not all, of the “bad” states are shown in red.

Any sequence of transitions is called a sentence. A sentence is accepted in an automaton if it takes us from an initial state to an accepting state. An automaton defines a language of sentences that get accepted. In our example sentences are over the alphabet \{-, w, g, c\} and the accepted sentences represent solutions to the puzzle. We can see in Figure 2 that the following sentences are among the accepted ones:

- g-wgc-b
- g-cgw-b
- g-egggw-b.

We can summarize that the automaton in the figure defines the language of all solutions of the river crossing puzzle.
Figure 2: Automaton for the river crossing problem.
The choice of states and transitions is often a modeling decision. For example, we could put more information in the states and represent also situations when the boat is just crossing the river. Intuitively this would not give us more solutions to the puzzle. In general, we prefer to have the smallest number of state such that the automaton contains enough information to solve the given problem.

Let us consider another example of an automaton. Suppose we are interested if a binary number is divisible by 3. To solve this task, we construct an automaton that reads a binary number from left to right and ends in the accepting state if the number is divisible by 3.

\[ \text{0101101011} \cdots \]
\[ \text{read} \]

The automaton, presented in Figure 3, has three state that represent the remainder of the number that has been read so far: rem 0, rem 1 and rem 2. If a binary number has been fully read and the automaton is in state rem 0, then the number is divisible by 3, therefore this state is accepting. The language defined by this automaton is the set of all binary number divisible by 3.

![Figure 3: Automaton for division by 3 problem.]

This automaton can be written down as a formal system that defines the same language:

\[
\begin{align*}
0 : \text{rm 0} & \quad 1 : \text{rm 1} \\
\text{x} : \text{rm 0} & \quad \text{x} : \text{rm 0} \\
\text{x0} : \text{rm 0} & \quad \text{x1} : \text{rm 1} \\
\text{x} : \text{rm 1} & \quad \text{x} : \text{rm 1} \\
\text{x0} : \text{rm 2} & \quad \text{x1} : \text{rm 0} \\
\text{x} : \text{rm 2} & \quad \text{x} : \text{rm 2} \\
\text{x0} : \text{rm 1} & \quad \text{x1} : \text{rm 2}
\end{align*}
\]

In this system, the states of the automaton became types of expressions. The rules in the first row tell us what the remainder is after reading the first symbol. The rules in other rows correspond to transitions in the automaton. In general, every automaton with a finite number of states can be written down as a formal system that defines the same language. The converse is not true – there are formal systems that cannot be converted to a finite automaton.
2 Union and intersection of languages

Consider two sets $A$ and $B$, for example two languages, which are set of sentences. The union of $A$ and $B$, denoted by $A \cup B$, is the set of all elements that are in $A$ or in $B$. The intersection of $A$ and $B$, written as $A \cap B$, is the set of all elements that are in $A$ and in $B$.

(a) Set union.
(b) Set intersection.

Figure 4: Set operations.

Union and intersection can be written down as rules in a formal system

$$
\frac{x : A}{x : A \cup B} \quad \frac{x : B}{x : A \cup B} \quad \frac{x : A}{x : A \cap B} \quad \frac{x : B}{x : A \cap B}
$$

The two rules on the left say that if $x$ belongs to type $A$ (respectively $B$) then it also belongs to type $A \cup B$. The rule on the right states that if $x$ belong both to type $A$ and $B$, then it also belongs to type $A \cap B$.

Suppose that we have two automata that define languages $A$ and $B$ and we are interested in an automaton that defines the intersection of their languages, i.e. $A \cap B$. We will show an example how such automaton can be constructed systematically. First, we need a second automaton that works on binary numbers. The automaton in Figure 3 defines binary numbers that have an even number of 1 symbols.

We use symbol $M_d$ for the automaton in Figure 3 and $M_e$ for the automaton in Figure 5. Let $L_e$ and $L_d$ be the languages defined by $M_e$ and $M_d$, respectively. To obtain a automaton that defines language $L_e \cap L_d$ we construct the product automaton of $M_e$ and $M_d$. This product is shown in Figure 6.

The idea of the product automaton is that it will simulate $M_e$ and $M_d$ at the same time. In other words it will check if a binary number has an even number of 1s and is divisible by 3. Every state in the product is pair $(s_1, s_2)$, such that $s_1$ is a state from $M_e$ and $s_2$ is a state from $M_d$. The initial state of the product automaton is the state that represents the initial states of $M_e$ and $M_d$. The transitions in the product correspond to the transitions in $M_e$ and $M_d$. More precisely, if in $M_e$ (respectively $M_d$) there is a transition $t$ from state $s_1$ to state $s'_1$, then in the
product automaton for every $s_2$ and $s'_2$ from $M_d$ ($M_e$ resp.) there is a transition $t$ from state $(s_1, s_2)$ to state $(s'_1, s'_2)$.

We are interested in intersection of languages $L_e$ and $L_d$, so the product automaton should be in a accepting state after reading a binary string only if $M_e$ and $M_d$ would be in their accepting states. Therefore, a state of the product automaton is accepting only if it represents accepting states of $M_e$ and $M_d$. Notice that the product of two automata, where one has $n$ states and the other has $m$ states, may have up to $m \cdot n$ states.

![Figure 6: Product of the automata in Figures 3 and 5 that accepts the intersection of their languages.](image)

Now suppose that we are interested in an automaton that defines $L_e \cup L_d$. It is enough to take the product automaton and add more accepting states. In this case, a state of the product automaton is accepting if it represents an accepting state of $M_e$ or of $M_d$ (see Figure 7).

![Figure 7: Product of the automata in Figures 3 and 5 that accepts the union of their languages.](image)
3 Nondeterminism

Automata we have seen so far are deterministic, which means that they are always in exactly one state. Now we will present nondeterministic automata, which can be in several states at the same time.

In a deterministic automaton for any state and any transition only one successor is available. More formally, if $Q$ is the set of states and $\Sigma$ is the alphabet of transitions, then successor state is given by a transition function $\delta$

$$\delta : Q \times \Sigma \rightarrow Q$$

In a nondeterministic automaton a state may have several successors for the same transitions. This means that $\delta$ now becomes a transition relation

$$\delta \subseteq Q \times \Sigma \times Q$$

Equivalently, we may write $\delta$ as a function from states and transitions to the set of states that are successors:

$$\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$$

where $\mathcal{P}(Q)$ is the power set of $Q$, which is the set of all subsets of $Q$. Deterministic automata are a special cases of nondeterministic automata that for every transitions always have at most one successor.

Imagine that we want an automaton that defines the set of binary numbers that have at least three digits and the 3rd digit from the end is 1. A deterministic automaton would need to remember the last three digits that have been read. Every digit can have two values, so this automaton would require at least $2 \cdot 2 \cdot 2 = 8$ states.

A nondeterministic automaton, like the one in Figure 8, can solve the same task with only four states. From state $a$ there are two successors for transition 1: state $a$ and state $b$. Choosing $b$ corresponds to making a guess that the 1 we have just read is going to be the 3rd digit from the end. Similarly, staying in $a$ corresponds to a guess that this digit is not going to be the 3rd last digit. Nondeterministic automata can try both choices and check later if some of them leads to an accepting state. In our example if the automaton is $a$ and reads 1, then it may go to $a$ and $b$ at the same time. If it is in $b$ and indeed there are only two more digits, then the automaton will end up in the accepting state $d$. However, if it is in $b$ there are fewer or more digits than two, then state $d$ will not be reached – this corresponds to checking that our guess was wrong.

![Figure 8: Nondeterministic automaton for the 3rd digit problem.](image)

A nondeterministic automaton can be converted to a deterministic one by a method called subset construction. The crux of this method is to represent any set of states in the nondeterministic automaton by a single state in the deterministic one. Figure 9 shows the subset construction for the nondeterministic automaton in Figure 8. Every states has a single successor for any transition, so this automaton is clearly deterministic. To see how the these two automata correspond to each other, suppose that we start in their initial states – which are $a$
and \( \{a\} \). After reading 1 the nondeterministic automaton is in states \( a \) and \( b \) at the same time, while the deterministic automaton is in state \( \{a, b\} \). For any input sequence the deterministic automaton is in a state that represents the set of states the nondeterministic automaton would be in after reading the same input.

![Subset construction for the nondeterministic automaton in Figure 8](image)

Above we said that a deterministic automaton would need to remember the last three digits to solve the 3rd digit problem. We can notice that the states of the automaton in Figure 9 actually encode the last three digits. The automaton in Figure 10 shows this encoding explicitly.

Any finite nondeterministic automaton can be converted by subset construction to a deterministic finite automaton. This implies that for finite automata nondeterminism does not add computational power. Nevertheless, a deterministic automaton may need many more states than a nondeterministic to define the same language.

In more expressive models of computation nondeterminism could make a difference. All computers that we know are deterministic, but we can imagine a hypothetical nondeterministic computer. It is unknown whether every problem that can be solved by a nondeterministic computer in polynomial time could also be solved in polynomial time by a deterministic computer. This problem is commonly known as \( P \) versus \( NP \) and it remains one of the biggest open question of computer science.
Figure 10: Automaton from Figure 9 with states encoded in binary.