1 Automata (count’d)

1.1 Nondeterminism

Automata we have seen so far are deterministic, which means that for every input exactly one transition is available from the current state. Now we will present nondeterministic automata, where several transitions may be possible.

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 1, 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 either \( a \) or \( b \). If \( b \) is chosen then two more digists have to follow to get to an accepting state.

The automaton represents the word for which a number of valid choices exists so that we end up in an accepting state. This is also known as guess-and-checking. The automaton can guess when the 3rd last digit is reached, but it has to check its guess.

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 2 shows the subset
construction for the nondeterministic automaton in Figure 1. 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 state $a$ or $b$, while the deterministic automaton is in state $\{a, b\}$, representing both possible choices of the non-deterministic automaton. For any input sequence the deterministic automaton is in a state that represents the set of states the nondeterministic automaton could be in after reading the same input.

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 2 actually encode the last three digits. The automaton in Figure 3 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.

1.2 Automata model discrete dynamical systems

Automata are powerful tools for modeling discrete dynamical systems. To know whether the system one wants to model is comprised in such class one must first clearly identify and define

- what are the states and
- what are the transitions.

We will introduce this concept with the river crossing puzzle of *The Wolf, the Goat and the Cabbage*. 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 the 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.

To solve this problem we formalize it as an automaton that is shown as a graph in Figure 3. 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.

As mentioned in the last lecture any sequence of transitions is called a *sentence*. 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 5 that the following sentences are among the accepted ones:

- g-wgc-b
- g-cgw-b
- g-cgggw-b.

We can summarize that the automaton in the figure defines the language of all solutions of the river crossing puzzle.
Figure 5: 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.

2 Reactive Modules

We saw that the concept of non-determinism in the context of finite automata does not increase the expressive power of the language defined. Reactive modules define a language used to describe state transition systems, and become particularly useful when the corresponding automaton is too complicated to be depicted through a diagram, or when multiple computing entities coexist in a system, and communicate through the exchange of signals.

The notion of reactive modules is introduced through the following example. Consider a system consisting of a train moving on rails forming a circular path, which intersects with a road. In the crossing, gate exists, which closes and opens the road according to the position of the train. To coordinate with the gate, the train signals the gate whenever it is approaching or leaving the intersection, to which the controller responds by taking the appropriate actions so that the gate is closed whenever the train is crossing, and open when the train is away.

We devise a mechanism for performing the above task by abstracting the system, focusing on the entities that are only relevant to the task. Initially, we partition the circular path into three areas: far, near, in, which define where the train is relative to the intersection. To coordinate with the gate, whenever the train is positioned near the intersection, it is issuing a signal approach, while when it have past the intersection, it is issuing a signal exit.

![Train Gate reactive modules](image)

Figure 6: The Train Gate reactive modules

When reactive modules are composed they end up in an automaton in which every state represent the state of every reactive module and transition are labelled by signals in case they are signaling transitions or unlabeled in case they are signal-free transitions. Later on in these notes we will give a more precise definition of reactive modules composition. As an example we show in Figure 7 the composition of the reactive modules of the Train Gate system. One can easily see that the composition of more complex reactive models could explode in a huge number of states. In general, with \( n \) modules \( M_1 \ldots M_n \) and \( |M| \) the number of states of the \( k \)-th module, we can have up to \( \prod_{k=1}^{n} |M_k| \) states. However, not all these states need to be reachable. Reactive modules allow us to define state transition system in a compact way,
dividing them in submodules. Forcing synchronization through signaling events allows us to reduce the state space of the composition. On such systems one can perform

- simulations,
- verification of
  - safety properties or
  - liveness properties.

Safety properties are of the kind “nothing bad happens”. Any violation happens in a finite number of steps. An example is whenever the train is in the gate the gate is closed. A counterexample would be any sequence of states from an initial state leading to a state in which the property does not hold. Trivially, the system that keeps the gate always closed is safe, but we say that it is not alive. Liveness properties are of the kind “something good eventually happens”. They are disproved by infinite sequences of steps. An example is whenever the gate is closed it will eventually open. It would be violated by a (reachable) loop of recurrent states in which the gate remains closed.

### 2.1 Textual Language For Reactive Modules

The following code defines the reactive module `train`, consisting of its possible state variables, together with input and output signals, as well as a description of its computation.

```plaintext
module Train
    state  x : {far, near, in}
    output approach!, exit!
    initially x = far
    update
        [] x = far approach! -> x = near
        [] x = near -> x = in
        [] x = in exit! -> x = far

```

Every line in the update section is a guarded command of the form $X \stackrel{U,Y!}{\rightarrow} Z$ (here no $U?$ is present), where $X$ is a guard (a formula over state variables) and $U?$ is a subset of incoming signals, $Y!$ a subset of outgoing signals and $Z$ denotes the update to the module state. We next define the module `gate`, which describes the behavior of the gate according to the received signals.
module Gate
  state \( y \): \{open, closed\}
  input approach?, exit?
  initially \( y = \text{open} \)
  update
  \[
  \begin{align*}
  &\, y = \text{open} \xrightarrow{\text{approach?}} y = \text{closed} \\
  &\, y = \text{closed} \xrightarrow{\text{exit?}} y = \text{open}
  \end{align*}
  \]

Note that all possible transitions need to be in the update section. The module must never get stuck because no guard is fulfilled. To ensure the module never gets stuck we can add a “do nothing” transition, equivalent to “\(\emptyset \rightarrow \text{true} \)”. If several updates are possible a non-deterministic choice is made. However, transitions involving a signal have priority.

The complete language of Simple Reactive Modules (SRM) is defined as follows:

Syntax A module consists of
- a set \( A \) of input events,
- a set \( B \) of output events,
- a set \( X \) of state variables,
- an initial condition on the state variables,
- a guarded command for each input event,
- a guarded command for each output event,
- a complete guarded command called update command.

A guarded command is a set of guarded actions. A guarded action consists of a guard, which is an enabling condition on the state variables, and an action, which is a modification of the state variables. A guarded command is complete if in every state at least one guard is true. A system is a set of modules whose output events are disjoint.

Semantics A state for a system consists of a state for every module. A state for a module is a function that maps every state variable to a value. A guarded command \( g \) changes module state \( q \) to module state \( q' \) if there exists a guarded action in \( g \) whose guard is true in \( q \) and whose action modifies \( q \) to \( q' \). Module state \( q' \) is an e-successor of module state \( q \) if \( e \) is an input or output event and the guarded command for \( e \) changes \( q \) to \( q' \). Module state \( q' \) is an update successor of module state \( q \) if the update command changes \( q \) to \( q' \). System state \( s' \) is an update successor of system state \( s \) if every module state in \( s' \) is an update successor of the respective module state in \( s \). System state \( s' \) is an e-successor of system state \( s \) if for every module \( M \), if \( e \) is an input or output event of \( M \), then the \( M \)-state in \( s' \) is an e-successor of the \( M \)-state in \( s \), else the \( M \)-state in \( s' \) is an update successor of the \( M \)-state in \( s \).