1 Reactive Modules

In the last lecture we introduced finite automata, together with some set operations between the languages they define, as well as the concept of non-determinism, which, in this context, 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, a gate exists, which operates under the presence of a controller. To coordinate with the controller, the train signals the controller 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 four areas: away, near, in, past, which define where the train is relative to the intersection. To coordinate with the controller, whenever the train is positioned near the intersection, it is issuing a signal approach, while when it is past the intersection, it is issuing a signal leave. 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.

module Train
  state: loc : {away, near, in, past}
  output: approach!, leave!
  initially: loc = away
  update:
   loc = away → approach! | loc = near
   loc = near → loc = in
   loc = in → loc = past
   loc = past → leave! | loc = away

Every line in the update section is a guarded command of the form \( X \rightarrow Y!|Z \) (here no \( U？ \) is present), where \( X \) is a guard (a formula over state variables) and \( U？ \) is a subset of incoming signals, and \( Y!, Z \) denote the corresponding actions to be taken, namely, issue some signals \( Y \) and change to state \( Z \). We next define the module controller, which describes the behavior of the controller according to the received signals.

module Controller
  state: gate : {open, closed}
  input: approach?, leave?
  initially: gate = open
  update:
   gate = open | approach? → gate = closed
   gate = closed | leave? → gate = open

Note that not all the possibilities are listed under the update section. Typically, we only state explicitly the ones that result in either changing the state of the module, or issuing some signals, while the rest are implicit “do nothing” transitions. So far we have just defined a system of two communicating automata (Fig. 1). We use the notation \( \text{Train} || \text{Controller} \) to refer to the whole system.

The trajectory of the system \( \text{Train} || \text{Controller} \) is a legal sequence of values for all the state variables, input and output signals of the system, such that consecutive entries follow legally from the specifications of
An invariant of some iterative mechanism is a property that remains unchanged through all the iterations of the mechanism. Typically, essential properties of systems are captured as invariants of all their (possibly infinite) trajectories. In our case, we are interested in the safety property which states that there does not exist a state of the system in which the train is in the intersection and the gate is open. Of course, such a property is now stated over the states of the respective variables \( \text{loc} \) and \( \text{gate} \) of the corresponding modules (Train and Controller), and is easily verified:

No trajectory exists where both \( \text{loc} = \text{in} \) and \( \text{gate} \neq \text{open} \).

Note that in our abstraction so far, we considered that the gate closes instantaneously. As in practice this normally is not the case, our model is not a realistic one. In a slightly more refined approach, the variable \( \text{gate} \) of the controller needs to transition through the state \( \text{closing} \), every time it is moving from \( \text{open} \) to \( \text{close} \):

\[
\begin{align*}
\text{module Controller2} \\
& \text{state: } \text{gate} : \{\text{open, closing, closed, opening}\} \\
& \text{input: } \text{approach?}, \text{ leave?} \\
& \text{initially: } \text{gate} = \text{open} \\
& \text{update:} \\
& \begin{cases} 
\text{gate} = \text{open} \mid \text{approach?} \rightarrow \text{gate} = \text{closing} \\
\text{gate} = \text{closing} \rightarrow \text{gate} = \text{closed} \\
\text{gate} = \text{closed} \mid \text{leave?} \rightarrow \text{gate} = \text{opening} \\
\text{gate} = \text{opening} \rightarrow \text{gate} = \text{open}
\end{cases}
\end{align*}
\]
Let us examine possible trajectories of the system \( Train \parallel Controller2 \). As one might have predicted, our safety property is no longer guaranteed! The reason is that there is no time restriction on the closing time of the gate, thus a signal from an approaching train might not have a rapid enough effect. The following trajectory is a witness of the violation of our safety property, and serves as a counterexample:

To deal with this timing issue, we equip our model with a third module, a clock. The task of the clock is to constantly tick, providing a logical time reference in our system. Then we can enforce events to happen in the system in a synchronized manner; for example, we can require that a train will transition from \( near \) to \( in \) only after 5 ticks from the time that it entered the \( near \) zone, while in the controller side, the can transition from \( closing \) to \( closed \) within the first 3 ticks of the clock.

**Table 2:** A bad execution of the \( Train \parallel Controller2 \) system, reaching a state where the train is in the intersection, but the gate is not closed.

<table>
<thead>
<tr>
<th>loc</th>
<th>approach</th>
<th>leave</th>
<th>gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>away</td>
<td>open</td>
<td></td>
<td></td>
</tr>
<tr>
<td>away</td>
<td>open</td>
<td></td>
<td></td>
</tr>
<tr>
<td>near</td>
<td>closing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>near</td>
<td>closing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>closing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2:** The \( Train2 \parallel Controller3 \parallel Clock \) system

**module Clock**
- output: \( tick! \)
- initially: –
- update:
  - \( [] \rightarrow tick! \)
2 Homework

Consider two trains moving on a circular path, one going clockwise and the other counter-clockwise. Normally there are two pairs of rails, so that there is no conflict between the two trains. However, the rails cross at some point, and the intersection is guarded by two signal lights, controller by a controller, whose job is to make sure that there is always at most one train in the intersection. Given that the actions of the trains are defined by the following two modules, provide the module of the controller such that the following three properties hold:
• If $loc_W = in$ then $loc_E \neq in$ and vice versa (i.e., the trains never cross the intersection simultaneously).

• If a train is waiting in the signal light, the other train cannot cross the intersection twice (liveness property)

• No deadlock occurs, i.e., at every possible trajectory of the system there is at least one successor that changes state.

module Train E
state: $loc_E : \{away, waiting, in\}$
external: $signal_W : \{green, red\}$
output: $approach_E!$, $leave_E!$
initially: $loc_E = away$
update:
\[
\begin{align*}
&\quad [ loc_E = away \rightarrow approach_E! | loc_E = waiting ] \\
&\quad [ loc_E = waiting \land signal_W = green \rightarrow loc_E = in ] \\
&\quad [ loc_E = in \rightarrow leave_E! | loc_E = away ]
\end{align*}
\]

module Train W
state: $loc_W : \{away, waiting, in\}$
external: $signal_E : \{green, red\}$
output: $approach_W!$, $leave_W!$
initially: $loc_W = away$
update:
\[
\begin{align*}
&\quad [ loc_W = away \rightarrow approach_W! | loc_W = waiting ] \\
&\quad [ loc_W = waiting \land signal_E = green \rightarrow loc_W = in ] \\
&\quad [ loc_W = in \rightarrow leave_W! | loc_W = away ]
\end{align*}
\]