EECS 149 MT1 Note Sheet

CH3 Discrete Dynamics:

inputs: *sigR*, *sigG*, *sigY* : pure
outputs: *pedestrian* : pure

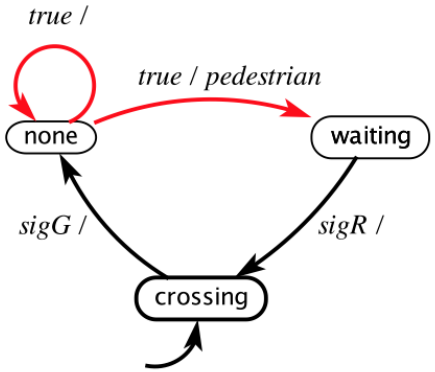


Figure 1: Non deterministic model of pedestrians that arrive at a crosswalk

Example 3.15: The FSM in Figure 3.11 can be formally represented as follows:

$$
\begin{aligned}
States &= \{\text{none}, \text{waiting}, \text{crossing}\} \\
Inputs &= (\{sigG, sigY, sigR\} \to \{present, absent\}) \\
Outputs &= (\{pedestrian\} \to \{present, absent\}) \\
initialStates &= \{\text{crossing}\}
\end{aligned}
$$

The update relation is given below:

$$
possibleUpdates(s,i) = \begin{cases} \{(\text{none}, absent)\} & \text{if } s = \text{crossing} \\ & \wedge\, i(sigG) = present \\ \{(\text{none}, absent), (\text{waiting}, present)\} & \text{if } s = \text{none} \\ \{(\text{crossing}, absent)\} & \text{if } s = \text{waiting} \\ & \wedge\, i(sigR) = present \\ \{(s, absent)\} & \text{otherwise} \end{cases} \tag{3.3}
$$

for all $s \in States$ and $i \in Inputs$. Note that an output valuation $o \in Outputs$ is a function of the form o: $\{pedestrian\} \to \{present, absent\}$. In (3.3), the sec-

Figure 2:The FSM formally represented

CH5 Composition of State Machines

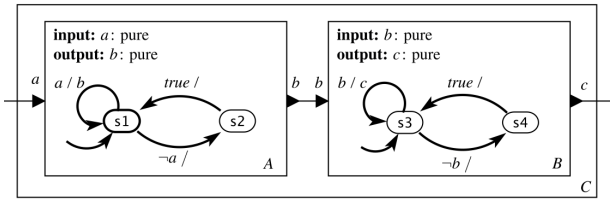


Figure 3: Cascade Composition of two FSMs

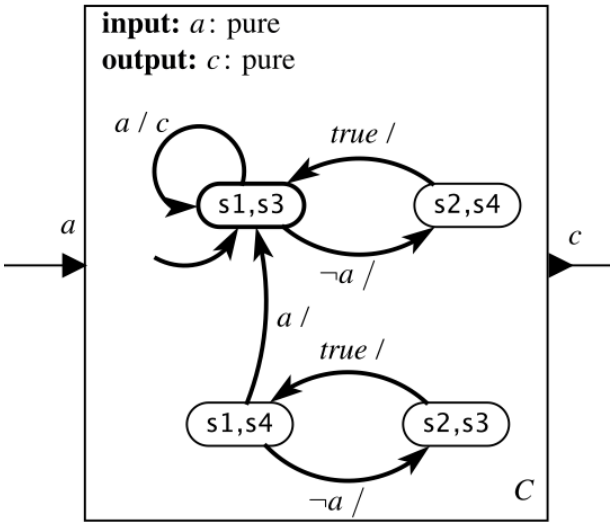


Figure 4: Semantics of the cascade composition, assuming synchronous composition

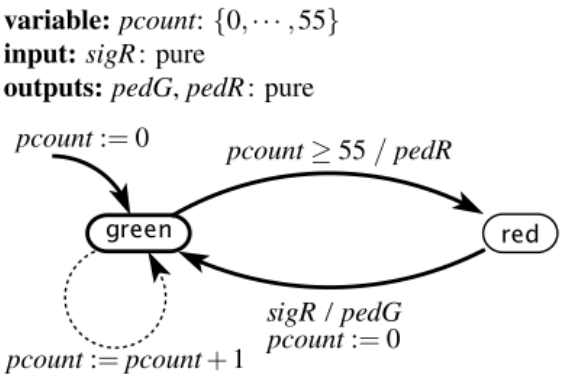


Figure 5: A model of a pedestrian crossing light, to be composed in a synchronous cascade composition with the traffic light model

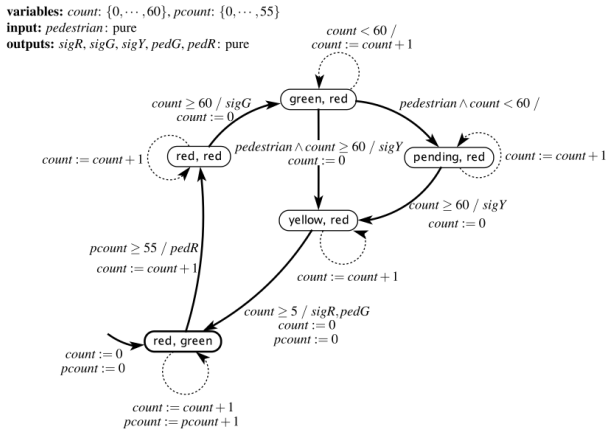


Figure 6: Semantics of a synchronous cascade composition of the traffic light model

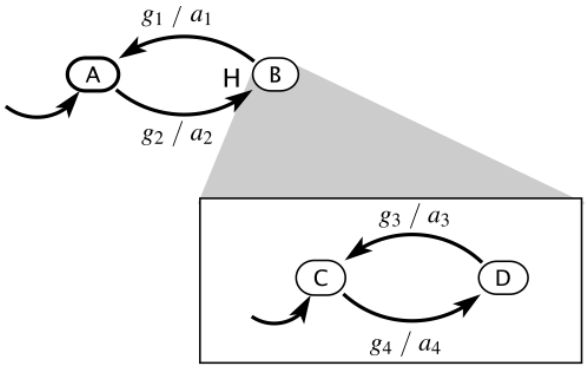


Figure 7: Variant of the hierarchical state machine of that has a history transition

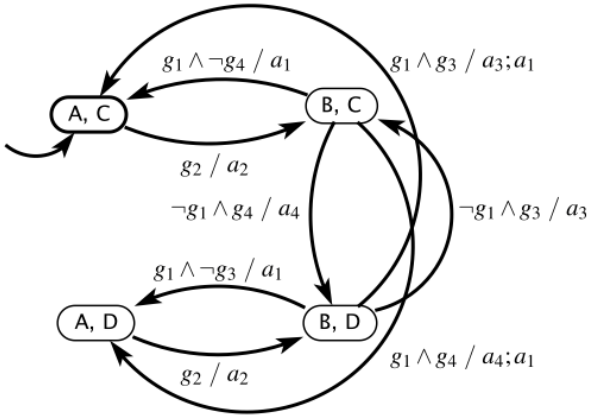


Figure 8: Semantics of the hierarchical state machine that has a history transition

CH7 Sensors and Actuators:

Digital sensors are unable to distinguish between two closely-spaced values of the physical quantity. The **precision** p of a sensor is the smallest absolute difference between two values of a physical quantity whose sensor readings are distinguishable. The **dynamic range** $D \in \mathbb{R}_+$ of a digital sensor is the ratio

$$D = \frac{H - L}{p},$$

where H and L are the limits of the range in (7.2). Dynamic range is usually measured in decibels (see sidebar on page 189), as follows:

$$D_{dB} = 20 \log_{10} \left(\frac{H - L}{p} \right). \tag{7.3}$$

Figure 9: Dynamic Range

CH9 Memory Architectures:

```
int a = 2;
void foo(int b, int* c) {
  ...
}
int main(void) {
  int d;
  int* e;
  d = ...;                    // Assign some value to d.
  e = malloc(sizeInBytes);    // Allocate memory for e.
  *e = ...;                   // Assign some value to e.
  foo(d, e);
  ...
}
```

In this program, the variable `a` is a **global variable** because it is declared outside any procedure definition. The compiler will assign it a fixed memory location. The variables `b` and `c` are **parameters**, which are allocated locations on the stack when the procedure `foo` is called (a compiler could also put them in registers rather than on the stack). The variables `d` and `e` are **automatic variables** or **local variables**. They are declared within the body of a procedure (in this case, `main`). The compiler will allocate space on the stack for them.

CH13 Invariants and Temporal Logic:

$x \wedge y$	True if x and y are both *present*.
$x \vee y$	True if either x or y is *present*.
$x = \textit{present} \wedge y = \textit{absent}$	True if x is *present* and y is *absent*.
$\neg y$	True if y is *absent*.
$\mathsf{a} \implies y$	True if whenever the FSM is in state a, the output y will be made present by the reaction

Note that if p_1 and p_2 are propositions, the proposition $p_1 \implies p_2$ is true if and only if $\neg p_2 \implies \neg p_1$. In other words, if we wish to establish that $p_1 \implies p_2$ is true, it is equally valid to establish that $\neg p_2 \implies \neg p_1$ is true. In logic, the latter expression is called the **contrapositive** of the former.

Note further that $p_1 \implies p_2$ is true if p_1 is false. This is easy to see by considering the contrapositive. The proposition $\neg p_2 \implies \neg p_1$ is true regardless of p_2 if $\neg p_1$ is true. Thus, another proposition that is equivalent to $p_1 \implies p_2$ is

The property $\mathbf{G}\phi$ (which is read as "**globally** ϕ") holds for a trace if ϕ holds for *every* suffix of that trace. (A **suffix** is a tail of a trace beginning with some reaction and including all subsequent reactions.)

The property $\mathbf{F}\phi$ (which is read as "**eventually** ϕ" or "**finally** ϕ") holds for a trace if ϕ holds for *some* suffix of the trace.

The property $\mathbf{X}\phi$ (which is read as "**next state** ϕ") holds for a trace $q_0, q_1, q_2, \ldots$ if and only if ϕ holds for the trace $q_1, q_2, q_3, \ldots$.

The property $\phi_1 \mathbf{U} \phi_2$ (which is read as "ϕ_1 **until** ϕ_2") holds for a trace if ϕ_2 holds for some suffix of that trace, and ϕ_1 holds until ϕ_2 becomes *true*.

Example 13.12: Consider the SpaceWire property:
"Whenever the reset signal is asserted the state machine shall move immediately to the ErrorReset *state and remain there until the reset signal is de-asserted."*

Let p be *true* when the reset signal is asserted, and q be true when the state of the FSM is *ErrorReset*. Then, the above English property is formalized in LTL as:

$$\mathbf{G}\,(p \implies \mathbf{X}(q\,\mathbf{U}\,\neg p))\,.$$

In the above formalization, we have interpreted "immediately" to mean that the state changes to *ErrorReset* in the very next time step. Moreover, the above LTL

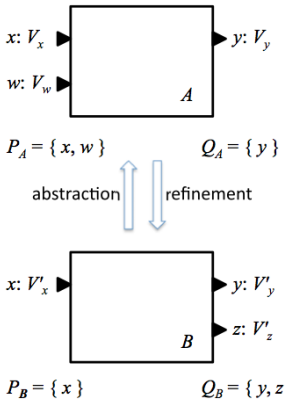
The following LTL formulas express commonly useful properties.

(a) *Infinitely many occurrences:* This property is of the form $\mathbf{G}\,\mathbf{F}p$, meaning that it is always the case that p is *true* eventually. Put another way, this means that p is true **infinitely often**.

(b) *Steady-state property:* This property is of the form $\mathbf{F}\,\mathbf{G}p$, read as "from some point in the future, p holds at all times." This represents a steady-state property, indicating that after some point in time, the system reaches a **steady state** in which p is always *true*.

(c) *Request-response property:* The formula $\mathbf{G}\,(p \implies \mathbf{F}q)$ can be interpreted to mean that a request p will eventually produce a response q.

CH14 Equivalence and Refinement:



(1) $P_B \subseteq P_A$

(2) $Q_A \subseteq Q_B$

(3) $\forall\, p \in P_B, \quad V_p \subseteq V'_p$

(4) $\forall\, q \in Q_A, \quad V'_q \subseteq V_q$

1. The first constraint is that B does not require some input signal that the environment does not provide. If the input ports of B are given by the set P_B, then this is guaranteed by

$$P_B \subseteq P_A. \tag{14.1}$$

The ports of B are a subset of the ports of A. It is harmless for A to have more input ports than B, because if B replaces A in some environment, it can simply ignore any input signals that it does not need.

2. The second constraint is that B produces all the output signals that the environment may require. This is ensured by the constraint

$$Q_A \subseteq Q_B, \tag{14.2}$$

where Q_A is the set of output ports of A, and Q_B is the set of output ports of B. It is harmless for B to have additional output ports because an environment capable of working with A does not expect such outputs and hence can ignore them.

The remaining two constraints deal with the types of the ports. Let the type of an input port $p \in P_A$ be given by V_p. This means that an acceptable input value v on p satisfies $v \in V_p$. Let V'_p denote the type of an input port $p \in P_B$.

3. The third constraint is that if the environment provides a value $v \in V_p$ on an input port p that is acceptable to A, then if p is also an input port of B, then the value is also acceptable B; i.e., $v \in V'_p$. This constraint can be written compactly as follows,

$$\forall\, p \in P_B, \quad V_p \subseteq V'_p. \tag{14.3}$$

Let the type of an output port $q \in Q_A$ be V_q, and the type of the corresponding output port $q \in Q_B$ be V'_q.

4. The fourth constraint is that if B produces a value $v \in V'_q$ on an output port q, then if q is also an output port of A, then the value must be acceptable to any environment in which A can operate. In other words,

$$\forall\, q \in Q_A, \quad V'_q \subseteq V_q. \tag{14.4}$$

The four constraints of equations (14.1) through (14.4) are summarized in Figure 14.1. When these four constraints are satisfied, we say that B is a **type refinement** of A. If B

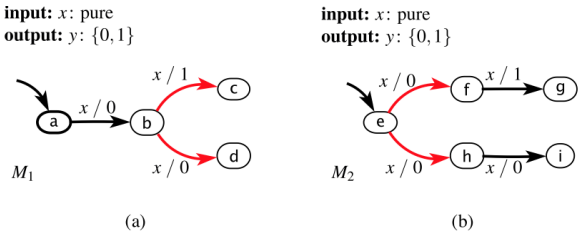


Figure 14.3: Two state machines that are language equivalent but where M_2 does not simulate M_1 (M_1 does simulate M_2).

Suppose we compose each of the two machines with its own copy of the environment that finds M_2 acceptable. In the first reaction where x is *present*, M_1 has no choice but to take the transition to state b and produce the output $y = 0$. However, M_2 must choose between f and h. Whichever choice it makes, M_2 matches the output $y = 0$ of M_1 but enters a state where it is no longer able to always match the outputs of M_1. If M_1 can observe the state of M_2 when making its choice, then in the second reaction where x is *present*, it can choose a transition that M_2 can *never* match. Such a policy for M_1 ensures that the behavior of M_1, given the same inputs, is never the same as the behavior of M_2. Hence, it is not safe to replace M_2 with M_1.

On the other hand, if M_1 is acceptable in some environment, is it safe for M_2 to replace M_1? What it means for M_1 to be acceptable in the environment is that whatever decisions it makes are acceptable. Thus, in the second reaction where x is *present*, both outputs $y = 1$ and $y = 0$ are acceptable. In this second reaction, M_2 has no choice but to produce one or the other these outputs, and it will inevitably transition to a state where it continues to match the outputs of M_1 (henceforth forever *absent*). Hence it is safe for M_2 to replace M_1.

(a) The simplest solution to achieve the desired effect is to trigger an ADC conversion upon each key-press event, if the ADC is available, for the specified key value, and to write to the file upon completion of the ADC conversion. Provide an implementation of the two ISRs to implement this.

SOLUTION:

```
1  void Key_ISR(){
2      axis val = keyboard_get_value();
3      if (!ADC_busy()){
4          ADC_trigger(val);
5      }
6  }
7  void ADC_ISR(){
8      write_ADC_result_to_file();
9  }
10 void main(){
11     ... setup and enable interrupts ...
12     while(1) sleep(10);
13 }
```