Introduction

In this exercise we are going to work with `boost::statechart`. Through an incremental approach we are going to get introduced to the basic building blocks which `boost::statechart` is comprised of.

In the directory `Radio` you will find the different harnesses.

Exercise 1 The vision

The big *picture* is the visionary diagram depicted in 1.1. The different exercises serve to be stepping stones ultimately leading to the implementation of this *UML Statemachine*.

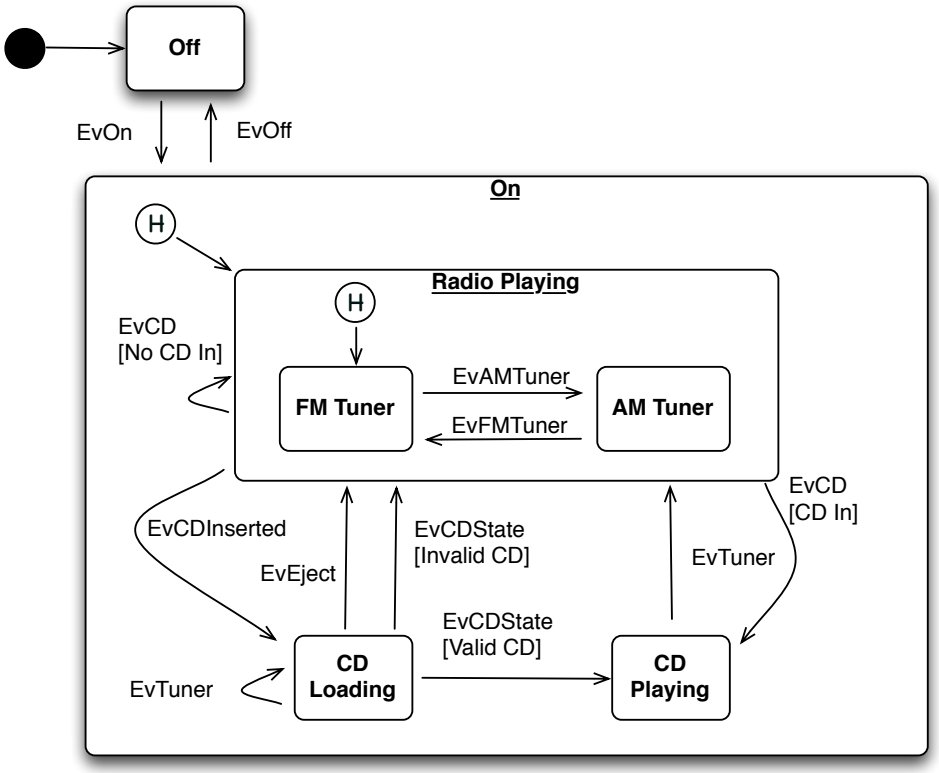

Figure 1.1: UML State diagram vision

For each named exercise you will find a sub directory with a similar name. In each of these a starting project has been placed and thus one you can utilize. The main focus of the exercise has been left out, meaning that most of the program has already been prepared, you job is to write the few missing lines of code. Obviously one could stick with the same project from start to finish but be aware that this will take more time.

Exercise 1.1 Simple states

In the directory `SimpleStates` you will find a C++ project where the below state machine is to be implemented.

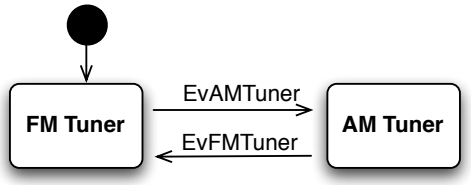

Figure 1.2: Simple states

Code snippets that are to be used in this exercise:

- `struct EvXXX sc::event<EvXXX> ;`
- `struct Machine : sc::state_machine<Machine, YYY> ;`
- `struct ZZZ : sc::simple_state<ZZZ, Machine> ;`
- `typedef sc::transition<???> reactions;`
- `process_event(EvXXX());`

Implement the whole state machine and do remember that it has to be initialized in function `main()` before it is ready to process any events.

Exercise 1.2 Multipe Hierarchical States

Next we are going to expand the solution such that we have multiple states in a hierarchy. The challenge is *not* to loose sight of the system design and thus how it should be implemented.

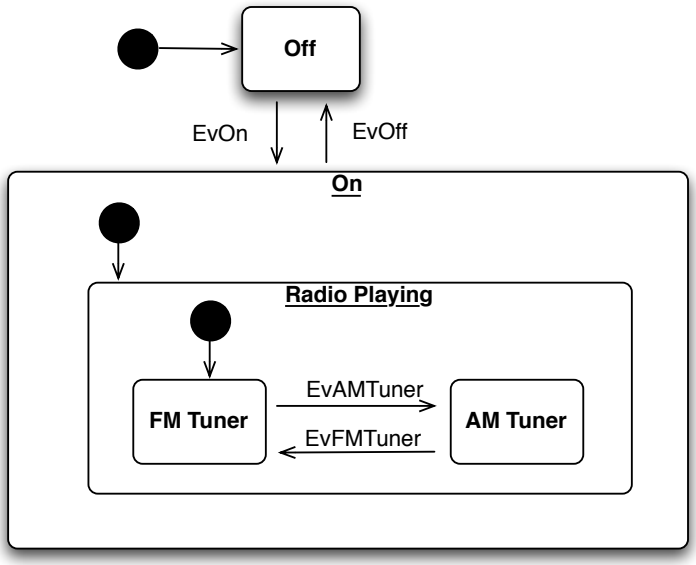

Figure 1.3: Multipe Heirarchical States

For this exercise we will be reusing already learned code but with an additional parameter:

- `struct ZZZ : sc::simple_state<ZZZ, Context, InnerState> ;`

The 3 template parameter *InnerState* is the initial nested state which is to be transitioned to when the parent state is chosen.

Implement the `struct On` and `struct RadioPlaying` in the prepared harness (See dir `Multiple-HierarchicalStates`).

Exercise 1.3 Guarded transition in a custom reaction

Sometimes transitions are only to be taken if certain criteria are met. In this exercise we add more states and a *custom reaction* in which it is decided (using a *guard*) which *transition* is to be taken.

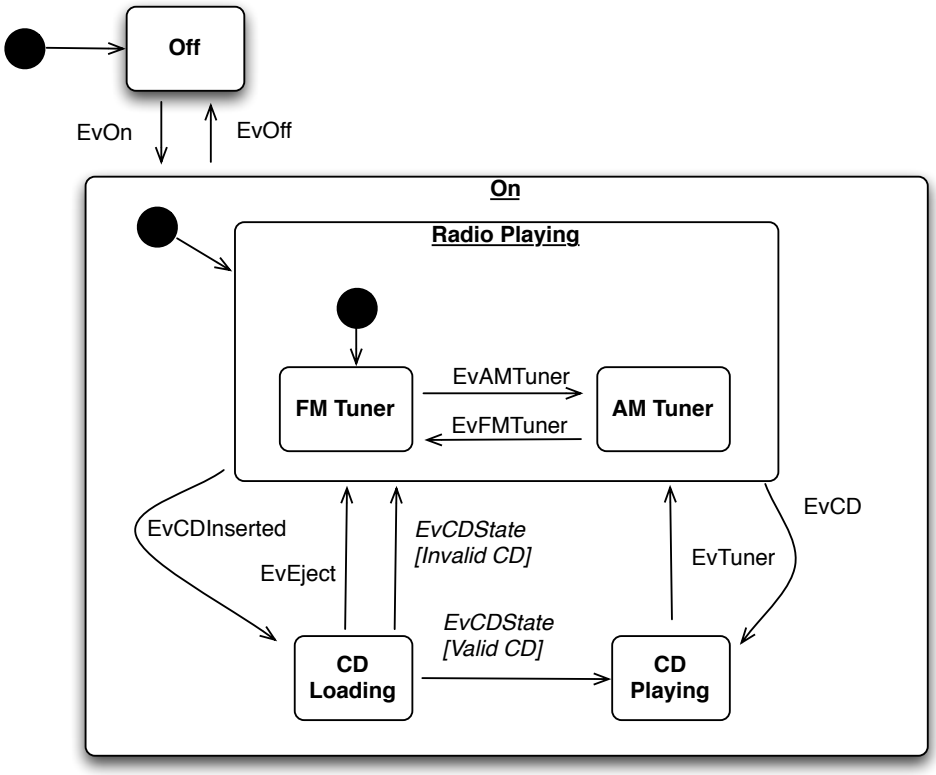

Figure 1.4: Guarded transition in a custom reaction

Additional code snippets that are to be used in this exercise:

- `boost::mpl::list<>`
- `sc::custom_reaction<???>`
- `transit<ZZZ>();`

Note that `boost::mpl::list<>` is used to `typedef` variable `reactions`, such that multiple reactions are valid.

Add an event `EvCDState` that in its constructor takes a `bool` and saves it in an internal data member. This data member is to be used as a guard in our custom reaction designating whether the state will change from *CDLoading* to state *RadioPlaying* or to state *CDPlaying*.

In this context it is evident that the above custom reaction/guard is to be implemented in `struct CDLoading`. Write the missing code and do remember to add a transition for the event `EvEject`.

See dir `GuardedTransInCustomReact` for harness.

Exercise 1.4 Instate reaction and state information

In responds to an event no state is necessarily changed. In this exercise a state variable from the state machine instance itself is used as a guard to determine whether a transition should be performed or not.

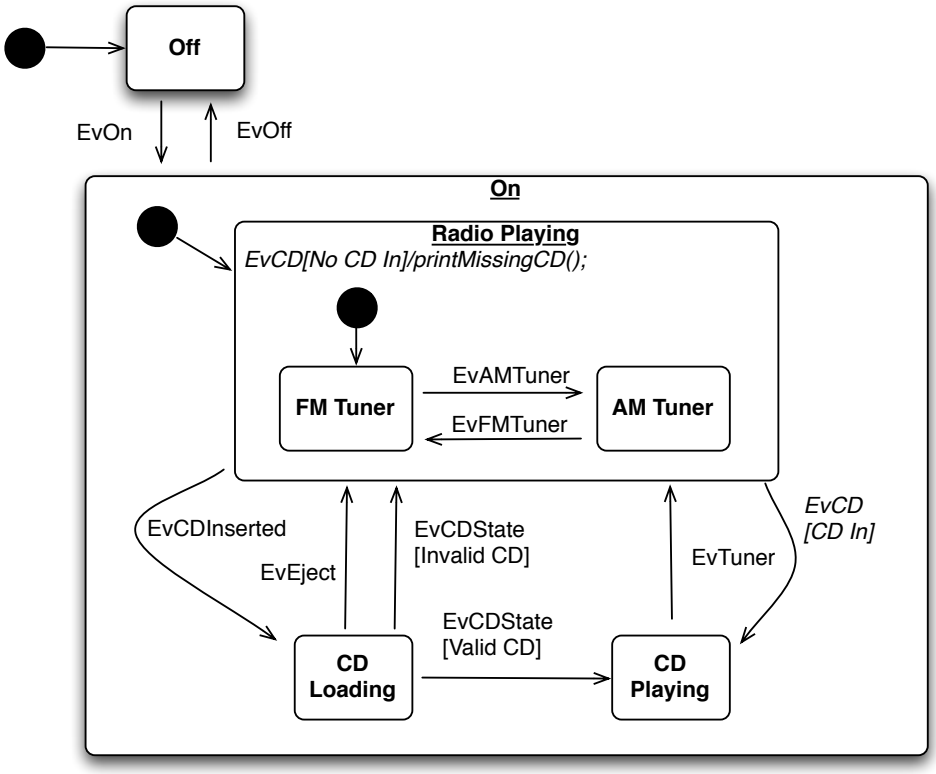

Figure 1.5: Instate reaction and state information

Additional code snippets that are to be used in this exercise:

- `context<???>().???`
- `discard_event();`

Class `RadioPlaying` is missing and its your job to implement it. As usual when implementing a state we need to add the type `reactions`. But furthermore a custom reaction for the event EvCD is needed. In this handler we will use the state variable `cdIn_` from class `Machine` to

determine whether this should be an *In State action* or that a change to state *CDPlaying* is to be performed.

See dir `InstateStateInformation` for harness.

Exercise 1.5 History

In certain situations remembering and changing to a previous chosen sub state in a given state hierarchy can prove to be an great enhancement.

For instance if one was listening to an FM channel, then chooses to listen to a CD for some time and then return to the radio. At this point one would expect that the original channel being listened to on the radio would the same one would be listening to when returning. For this to work some kind of memory or *history* of what was done earlier is required.

As can been seen in figure 1.6 the history notation has been added to produce the above described desired effect.

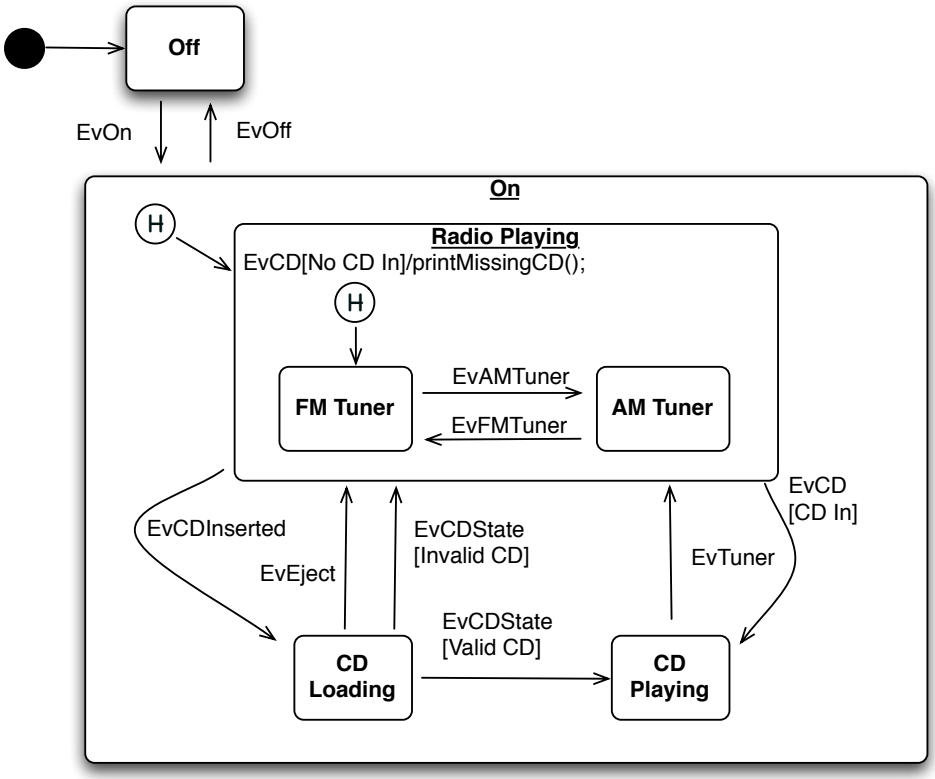

Figure 1.6: History

Additional code snippets that are to be used in this exercise:

- `shallow_history<???>`
- `has_shallow_history`

The existing code harness is identical to the solution from the previous exercise. Your job is to change this code such that is reflects UML state diagram in 1.6.

See dir `History` for harness.