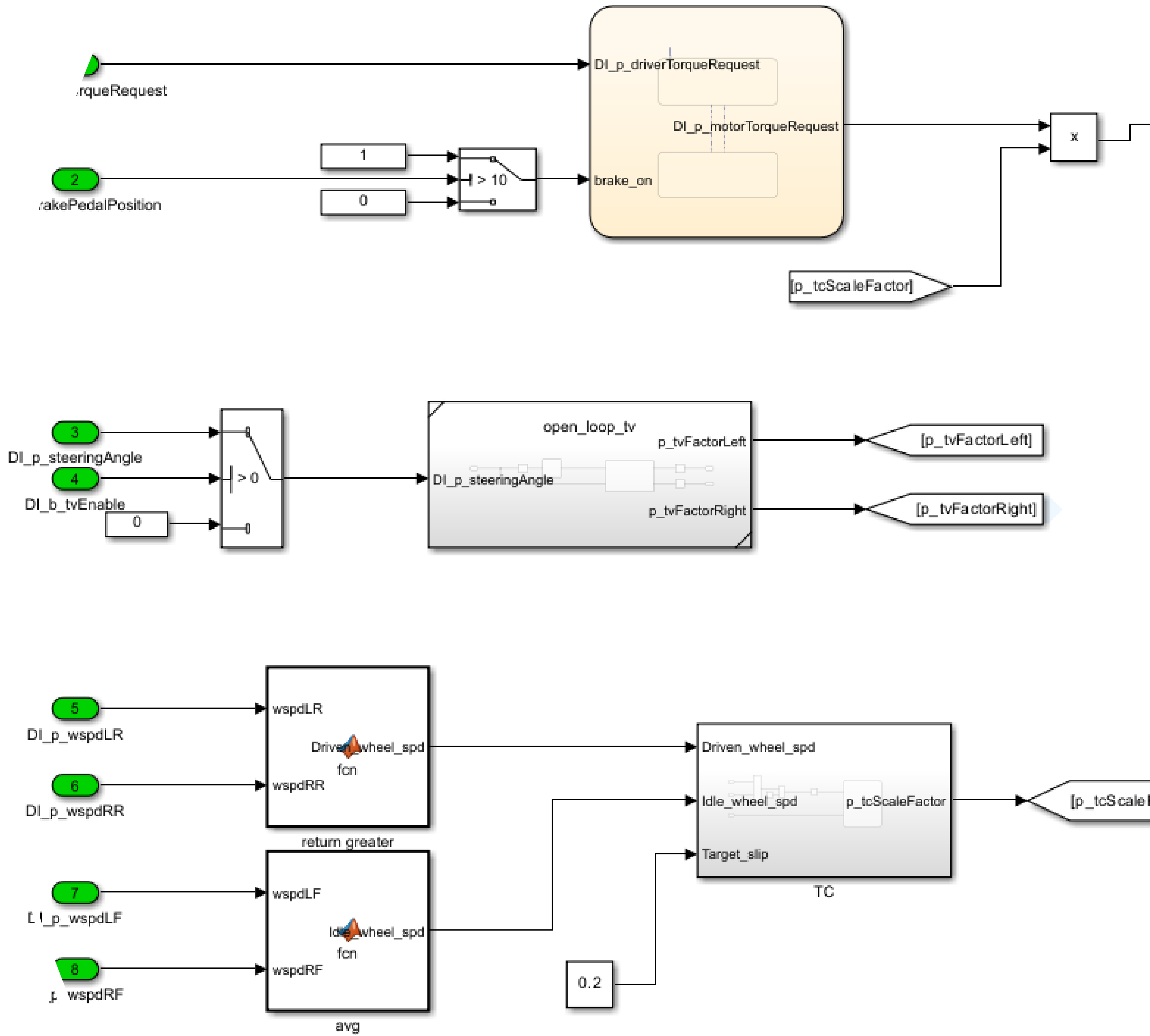
Simulink to C++

Luai, Manush, Tarun, Teghveer

What is the Simulink Model?

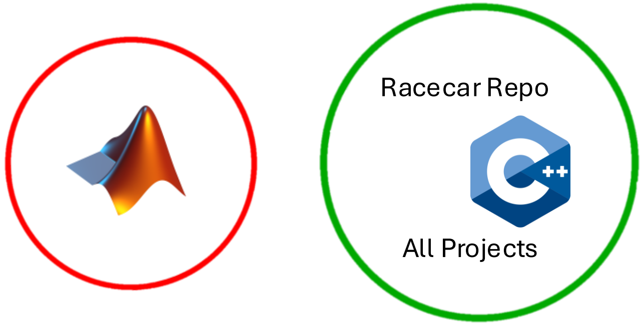
- Simulink → Graphical tool for modelling and simulating a dynamic system.
- controller.slx → Manages all systems on the racecar.
  - Battery monitor
  - Motor interface
  - Driver interface
  - Simple vehicle dynamics
  - Governor
- State machines → System behaviour as a set of states, transitions, and actions.
  - Easy to model
  - Modular
  - Scalable



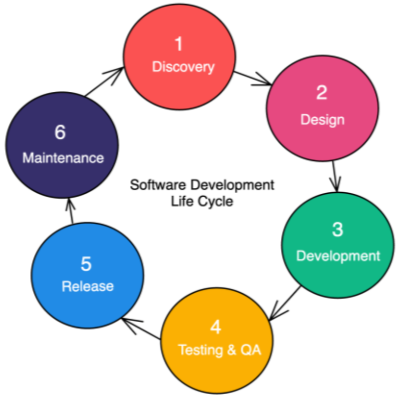
Why Convert it to C++?

1. Compared to C++, MATLAB is slow to compile with

2. Tighter integration with our racecar repo
   - Works better with other components needed

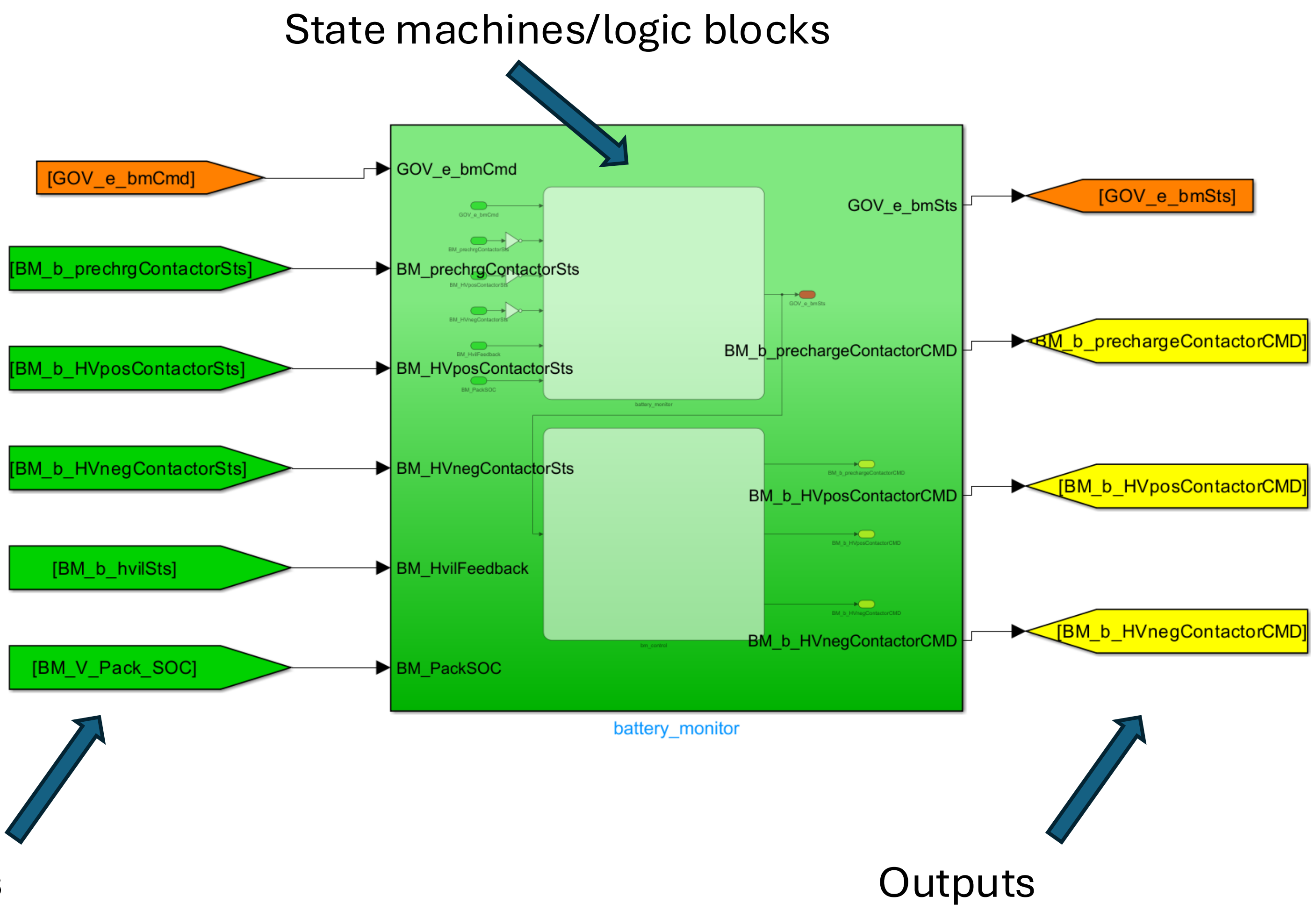


3. Better development cycle
   - Working with an already familiar language in the repo
   - Easier to make/run tests



Simulink blocks to C++ blocks – the design

- Simulink blocks had the following properties:
  - Set of inputs
  - Set of outputs
  - State machines/logic blocks to compute the logic



Design: Input/Output Structs

- Structs are made for the input and the output of the block
- This encapsulates values together
- Also allows to define a black box for the new design
    - If we provide this input, we will get an output in this structure
- Enums/other structs are made to be used as possible values of the input/output

```
struct Input {
    BmCmd cmd;
    ContactorState precharge_contactor_states;
    ContactorState hv_pos_contactor_states;
    ContactorState hv_neg_contactor_states;
    ContactorState hvil_status;
    float pack_soc;
};
```

```
struct Output {
    BmStatus status;
    ControlStatus control_status;
    ContactorCMD contactor;
};
```

```
enum class ControlStatus {
    STARTUP_CMD,
    CLOSE_HV_NEG,
    CLOSE_PRECHARGE,
    CLOSE_HV_POS,
    OPEN_PRECHARGE,
};

enum class BmCmd {
    INIT,
    HV_STARTUP,
    HV_SHUTDOWN,
};

enum class ContactorState : bool {
    OPEN = false,
    CLOSED = true,
};

struct ContactorCMD {
    ContactorState precharge;
    ContactorState hv_positive;
    ContactorState hv_negative;
};
```

Design: Class for the Block

- **Public Interface:**
  - Update method, you provide the input and the current time, and it provides an output
  - Think of it as the black box!
- **Private Interface:**
  - Transition method, which is used to transition from one state to another in the FSM
  - Other variables needed for the block

- time_ms tracks the time at which Update is run, and tracks how long we have been in one state for

```
struct Input {
    input1;
    input2;
    ...
}
```

```
struct Output {
    output1;
    output2;
    ...
}
```

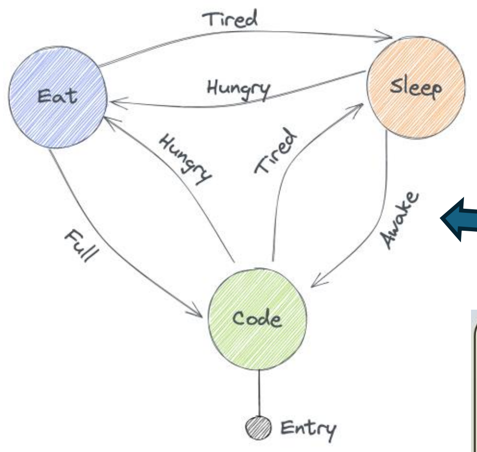
```
struct FsmStates {
    state1;
    state2;
    ...
}
```

```
class CppBlock {
public:
    Output Update(const Input& input, int time_ms);

private:
    FsmStates Transition(const Input& input, int time_ms);

    FsmStates state_;
    int state_start_time_;

    Any other variables needed!
};
```

Side Note: What is a Finite State Machine?

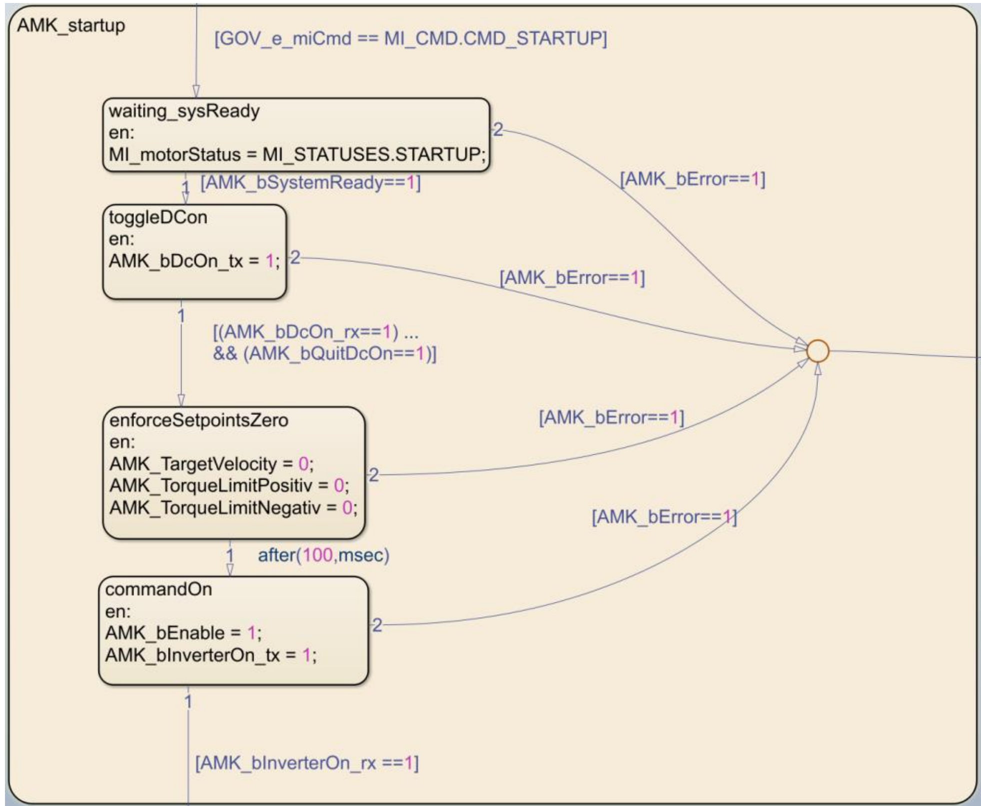
- Multiple states defined that each do something
- Each state has different conditions to transition to another state

- On each run of the FSM:
    1. Look at the current state you are in, which is stored from the previous run
    2. Run transition conditions for the state and transition to next state if it passes
    3. Execute the action for the new state
    4. Repeat for the next run!

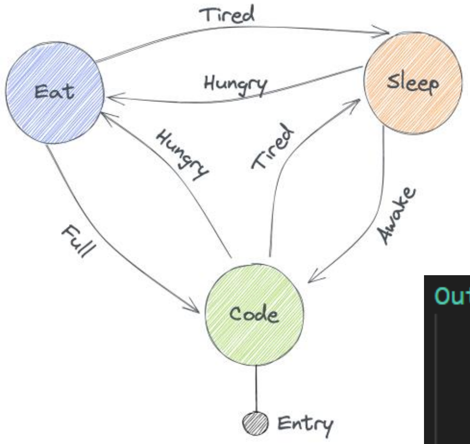


State

Transition Condition



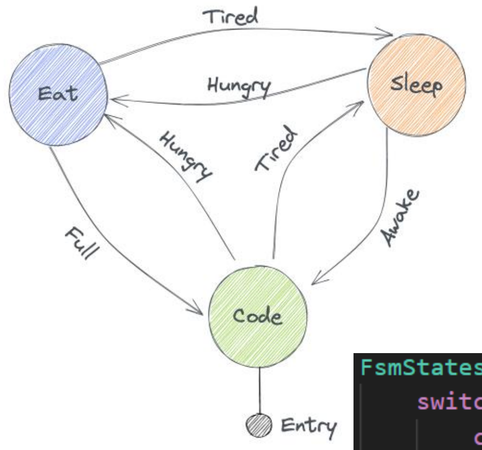
Design: Update Method



- Let’s continue with the eat/sleep example!
- Transition is first called to get the new state for the update
- A switch statement is used to set a case for each possible state in the FSM
- Define each case, and set the action that each state does
  - This would just be creating the output for that state
- Finally, return the output for that update run!

```cpp
Output CppBlock::Update(const Input& input, const int time_ms) {
    state_ = Transition(input, time_ms);

    Output output{};
    switch(state_) {
        case CODE:
            output = code();

            break;

        case SLEEP:
            output = sleep();

            break;

        case EAT:
            output = eat();

            break;
    }

    return output;
}
```

Design: Transition Method



- Used by the Update method to transition from one state to another
- Same switch statement idea, but now there are checks for each transition condition
- If a condition passes, switch the state to that new state!
- If no conditions pass, remain in the same state

```
FsmStates CppBlock::Transition(const Input& input, const int time_ms) {
    switch(state_) {
        case CODE:
            if (tired) state_ = SLEEP;
            else if (hungry) state_ = EAT;

            break;

        case SLEEP:
            if (awake) state_ = code;
            else if (hungry) state_ = EAT;

            break;

        case EAT:
            if (tired) state_ = SLEEP;
            else if (full) state_ = CODE;

            break;
    }

    return state_;
}
```

Following a Real Example: AMK Block

- Output mimic’s block

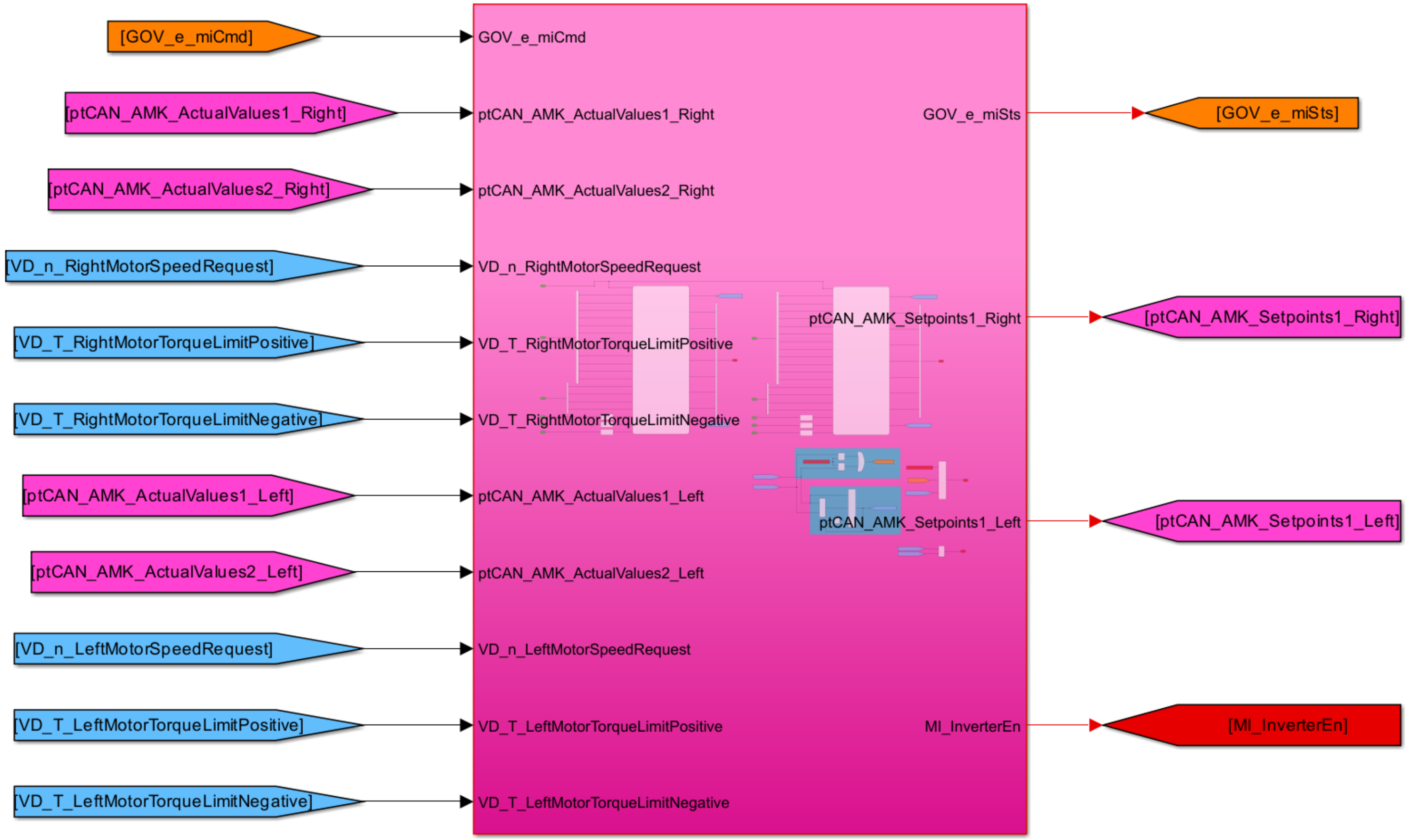
```
// AmkOutput struct, the output that is produced from the state machine
struct AmkOutput {
    MiStatus status;
    generated::can::AMK0_SetPoints1 left_setpoints;
    generated::can::AMK1_SetPoints1 right_setpoints;
    bool inverter_enable;
};
```

- Input mimic’s block

```
// AmkInput struct, the input that is fed into the state machine
struct AmkInput {
    MiCmd cmd;
    generated::can::AMK0_ActualValues1 left_actual1;
    generated::can::AMK1_ActualValues1 right_actual1;
    MotorInput left_motor_input;
    MotorInput right_motor_input;
};
```

- Class defines Update, Transition, etc

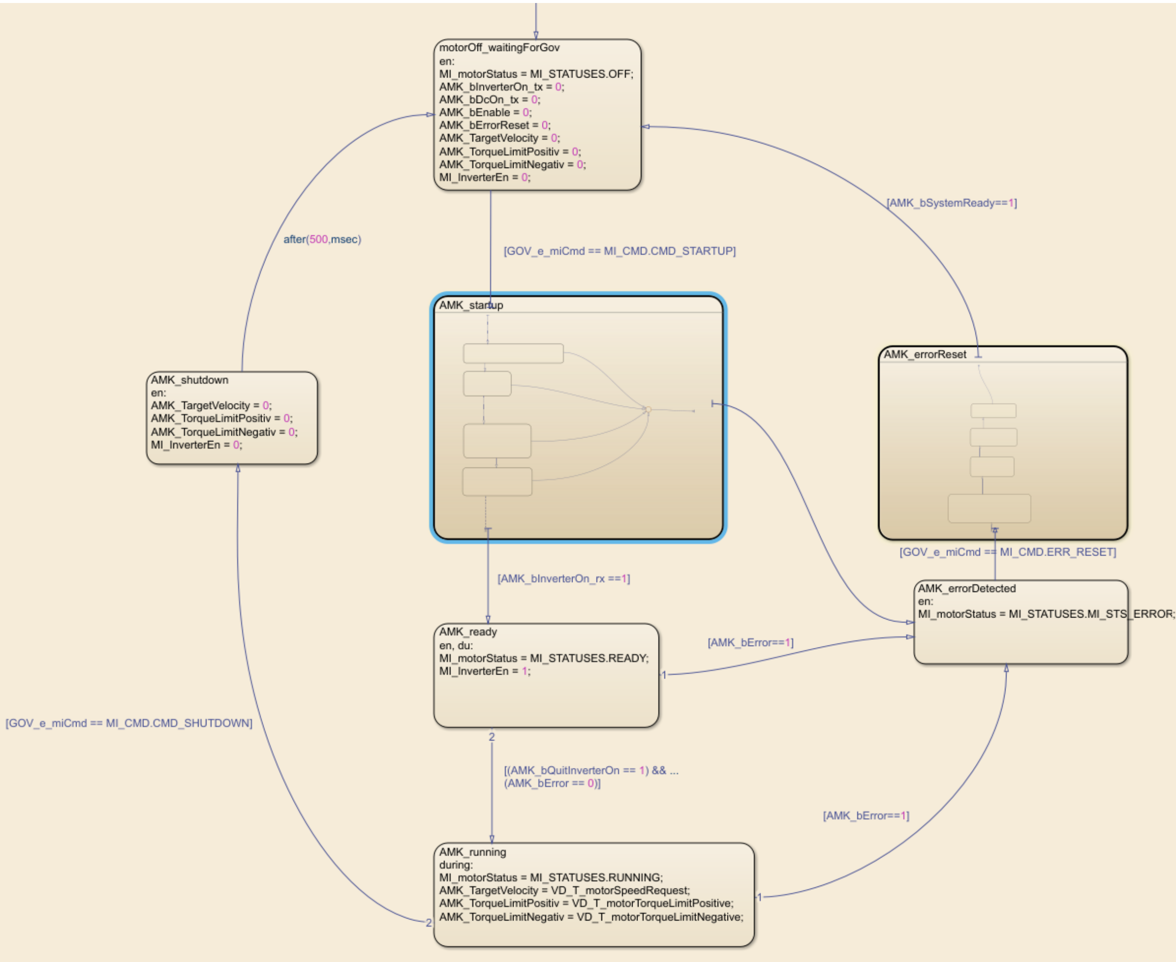
```
template <AmkActualValues1 V1, SetPoints SP>
class AmkManager {
public:
    AmkManager(
        AmkStates initial_amk_state = AmkStates::MOTOR_OFF_WAITING_FOR_GOV);

    UpdateMotorOutput<SP> UpdateMotor(const V1 val1,
                                      const MotorInput motor_input,
                                      const MiCmd cmd, int time_ms);

    AmkStates Transition(const V1 val1, const MiCmd cmd, const int time_ms);

private:
    AmkStates amk_state_;
    int amk_state_start_time_ = 0;
    UpdateMotorOutput<SP> output_{.status = MiStatus::OFF,
                                  .inverter_enable = false};
};
```



Following a Real Example: AMK Block

- Transition called
- Each state defined
- Each state's actions defined
- Output constructed and returned

```
template <AmkActualValues1 V1, SetPoints SP>
UpdateMotorOutput<SP> AmkManager<V1, SP>::UpdateMotor(
    const V1 val1, const MotorInput motor_input, const MiCmd cmd,
    const int time_ms) {
    using enum AmkStates;

    amk_state_ = Transition(val1, cmd, time_ms);
    switch (amk_state_) {
        case MOTOR_OFF_WAITING_FOR_GOV:
            output_.status = MiStatus::OFF;
            output_.inverter_enable = false;

            output_.setpoints.amk_b_inverter_on = false;
            output_.setpoints.amk_b_dc_on = false;
            output_.setpoints.amk_b_enable = false;
            output_.setpoints.amk_b_error_reset = false;
            output_.setpoints.amk__target_velocity = 0;
            output_.setpoints.amk__torque_limit_positiv = 0;
            output_.setpoints.amk__torque_limit_negativ = 0;

            break;

        case STARTUP_SYS_READY:
            output_.status = MiStatus::STARTUP;

            break;

        case STARTUP_TOGGLE_D_CON:
            output_.setpoints.amk_b_dc_on = true;

            break;

        case STARTUP_ENFORCE_SETPOINTS_ZERO:
            output_.setpoints.amk__target_velocity = 0;
            output_.setpoints.amk__torque_limit_positiv = 0;
            output_.setpoints.amk__torque_limit_negativ = 0;

            break;
```

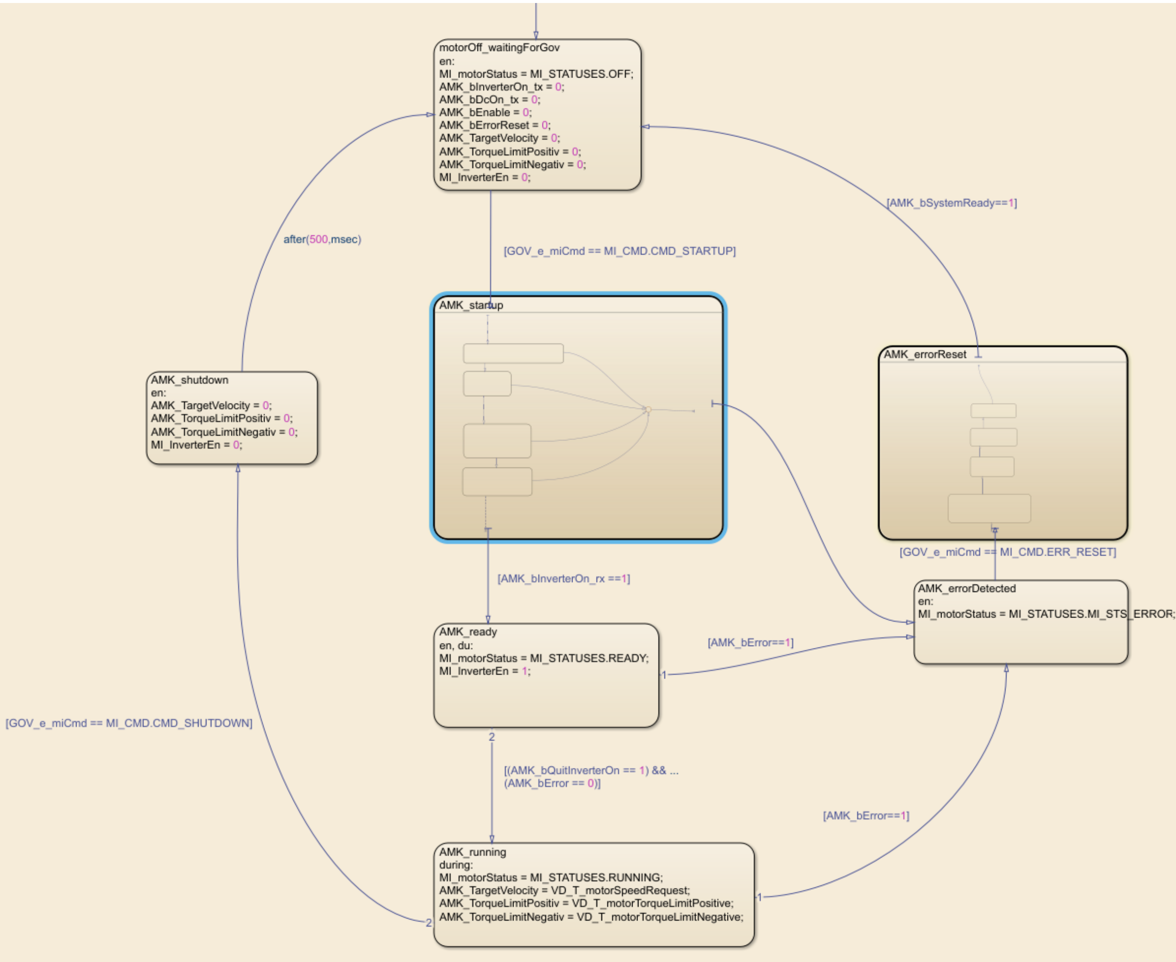


Following a Real Example: AMK Block

- Each state defined
- Each state's transitions defined
- New state returned

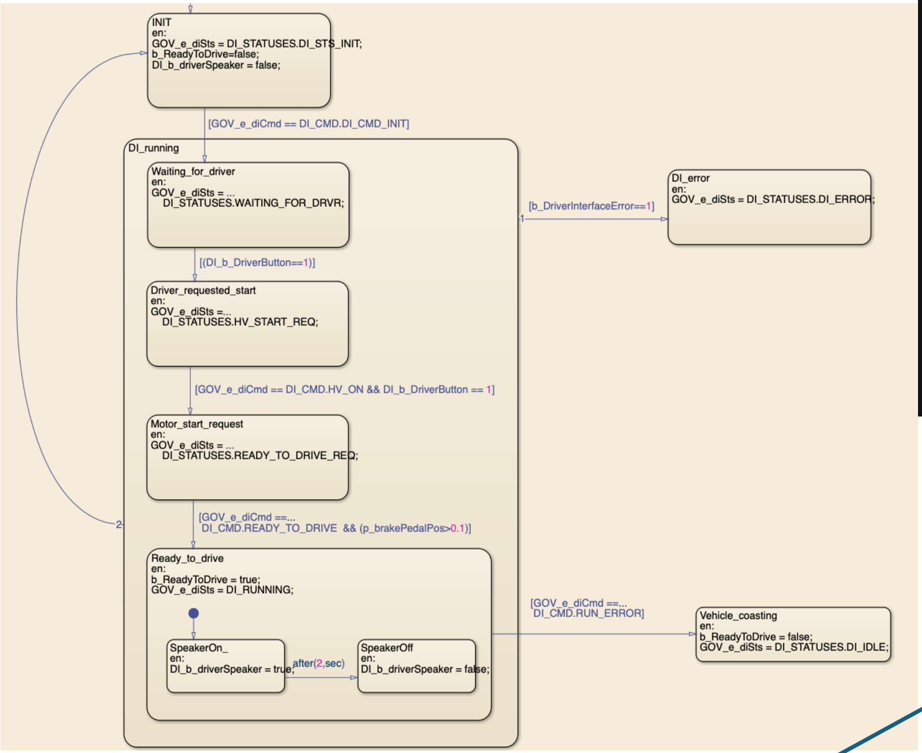
- Error transitions are the same, group them up at the top
- time_ms is used to find elapsed time and if state is ready to transition

```
AmkStates AmkManager<V1, SP>::Transition(const V1 val1, const MiCmd cmd,
                                         const int time_ms) {
    using enum AmkStates;

    // If any of these states have amk_b_error set, move to ERROR_DETECTED state
    if ((amk_state_ == STARTUP_SYS_READY ||
         amk_state_ == STARTUP_TOGGLE_D_CON ||
         amk_state_ == STARTUP_ENFORCE_SETPOINTS_ZERO ||
         amk_state_ == STARTUP_COMMAND_ON || amk_state_ == READY ||
         amk_state_ == RUNNING) &&
        val1.amk_b_error) {
        amk_state_start_time_ = time_ms;
        return ERROR_DETECTED;
    }

    AmkStates new_state = amk_state_;
    int elapsed_time = time_ms - amk_state_start_time_;

    switch (amk_state_) {
        case MOTOR_OFF_WAITING_FOR_GOV:
            if (cmd == MiCmd::STARTUP) {
                new_state = STARTUP_SYS_READY;
            }

            break;

        case STARTUP_SYS_READY:
            if (val1.amk_b_system_ready) {
                new_state = STARTUP_TOGGLE_D_CON;
            }

            break;

        case STARTUP_TOGGLE_D_CON:
            if (val1.amk_b_dc_on && val1.amk_b_quit_dc_on) {
                new_state = STARTUP_ENFORCE_SETPOINTS_ZERO;
            }

            break;

        case STARTUP_ENFORCE_SETPOINTS_ZERO:
            if (elapsed_time >= 100) {
                new_state = STARTUP_COMMAND_ON;
            }

            break;
```



Testing C++ Blocks

- Separate test.cc file running on cli
- State specific/Transition testing
- Error Handling
- Uniform structure amongst all blocks
- Important to test how a transition affects an output or vice versa
- Assert statements
- Perhaps use GoogleTest moving forwards?
  - Better framework, no crashes on failure, descriptive test names and can integrate with CMAKE




```cpp
void test_sequence() {

        out = fsm.Update(in, time_ms++);
        ASSERT_EQ(out.status, DiSts::WAITING_FOR_DRVR);
        ASSERT_FALSE(out.ready_to_drive);
        ASSERT_FALSE(out.speaker_enable);
    }

    { // Nothing should happen if button isn't pressed
        in.driver_button = false;
        out = fsm.Update(in, time_ms++);
        ASSERT_EQ(out.status, DiSts::WAITING_FOR_DRVR);
    }

    { // Start the motor only if the high-voltage is on and the driver presses
      // the button
        in.driver_button = true;
        out = fsm.Update(in, time_ms++);
        ASSERT_EQ(out.status, DiSts::HV_START_REQ);
        ASSERT_FALSE(out.ready_to_drive);
        ASSERT_FALSE(out.speaker_enable);

        out = fsm.Update(in, time_ms++);
        ASSERT_EQ(out.status, DiSts::HV_START_REQ);

        in.command = DiCmd::HV_ON;
        in.driver_button = false;
```

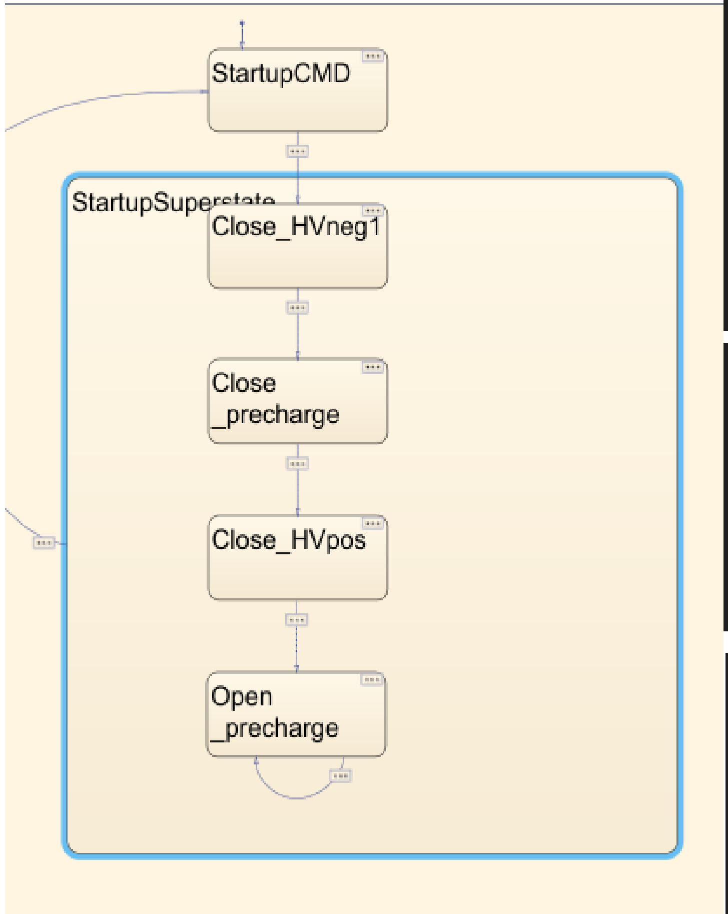
```cpp
#include <gtest/gtest.h>
#include "DriverInterface.h"

TEST(BrakeLightTest, ActivatesAboveThreshold) {
    DriverInterface di;
    di.brake_pedal_pos = 0.6;
    EXPECT_TRUE(di.brake_light_status());
}

TEST(BrakeLightTest, DoesNotActivateBelowThreshold) {
    DriverInterface di;
    di.brake_pedal_pos = 0.2;
    EXPECT_FALSE(di.brake_light_status());
}
```

Groups related tests

Describes individual test

Design: State Transition with std::optional

- std::optional is used to manage our FSM's, ensuring they are only accessed when valid.
- .has_value() checks prevent operations on uninitialized states
- If current_status_ is uninitialized, the system forces INIT to guarantee a defined starting state.
- .value() is used when a valid state is confirmed




```cpp
private:
    std::optional<BmStatus> TransitionStatus(const Input& input, int time_ms);
    std::optional<ControlStatus> TransitionControl(BmStatus status,
                                                   int time_ms);
    ContactorCMD SelectContactorCmd(ControlStatus bm_control_status_);

    // State machine variables (BmUpdate)
    std::optional<BmStatus> current_status_;
    std::optional<ControlStatus> bm_control_status_;
    int status_snapshot_time_ms_;
    int control_snapshot_time_ms_;
};
```

```cpp
std::optional<BmStatus> BatteryMonitor::TransitionStatus(const Input& input,
                                                         int time_ms) {
    using enum ContactorState;
    using enum BmStatus;

    if (!current_status_.has_value()) {
        return INIT;
    }
```

```cpp
BatteryMonitor::Output BatteryMonitor::Update(const Input& input, int time_ms) {
    auto new_transition = TransitionStatus(input, time_ms);

    if (new_transition.has_value()) {
        status_snapshot_time_ms_ = time_ms;
        current_status_ = new_transition.value();
    }

    auto new_control_transition =
        TransitionControl(current_status_.value(), time_ms);

    if (new_control_transition.has_value()) {
        control_snapshot_time_ms_ = time_ms;
        bm_control_status_ = new_control_transition.value();
    }

    ContactorCMD contactor_cmd = SelectContactorCmd(bm_control_status_.value());
```

Learnings/Reflection

- Luai – loved learning how to codify FSMs in C++
- Manush – I enjoyed learning C++, reading Simulink models and thinking about code that directly affects the driver and the car
- Tarun – Enjoyed learning about FSMs, integrating Simulink models into C++, using std::optional, and finding ways to simplify complex code.
- Teghveer – Integrating a time sensitive system was interesting.

Questions?