LLM, MDD and Micro-Modularity

Dirk Engel, info@engel-internet.de, November 26, 2023
https://github.com/dirkengel/

Introduction

In "Neither Human nor Machine", I highlighted that in software development, ChatGPT can serve as an effective sparring partner and a valuable tool, providing the desired results as long as trust is maintained. The principle "Trust, but verify" comes into play. Verification can be achieved by applying the divide-and-conquer principle, breaking down tasks into smaller, manageable pieces, see "Divide (along interfaces) and Conquer". This allows for a detailed review, understanding, improvement, or approval of the responses generated by the Large Language Model (LLM) system.

Particularly when dealing with recurrent software problems and aiming for a generic solution by describing the task's mechanics at a meta-level, micro-modularity can be employed. This involves breaking down tasks into smaller components to create high-quality code templates. These templates, multiplied by a generator, influence code quality in various places ("A Software Dependency Flip Book"). Therefore, special attention to these code parts is crucial.

In terms of quality attributes, we can address this by requesting idiomatic implementations from the LLM in various programming languages to enhance portability. Moreover, we can target performance efficiency by requesting different code variants for the same snippet. When discussing "different", we are not limited to comparing a few alternative implementations; the LLM should empower us to perform a kind of brute force analysis of as many variants as possible. In essence, developers and their sparring partner can collaboratively explore and analyze a broad spectrum of code variations.

Example

Simple State Machine

The example is a small state machine written in C++. Rather than starting with a blank piece of paper, I chose to specify in prose how I envision the states and transitions. Additionally, I aim to allow the injection of guard conditions as predicate functions, transition actions, and state entry and exit action code. To complete the picture, some test code is also desirable. After a few iterations, this provides enough input for ChatGPT to generate code that can serve as a good starting point. Here is the Source Code after several manual *editorial* changes:

```
// file: fsm.h -> framework code

#include <functional>

class State {
public:
  State() = default;
  ~State() = default;

  std::function<void()> entryAction {nullptr};
  std::function<void()> exitAction {nullptr};
};

class Transition {
public:
  Transition(State* from, State* to) : fromState(from), toState(to) {}
  ~Transition() = default;

  State* getFromState() const {return fromState;}
  State* getToState() const {return toState;}
```

```cpp

  std::function<bool()> condition {[] {return true;}}; // valid by default
  std::function<void()> action {nullptr};

private:
  State* fromState {nullptr};
  State* toState {nullptr};
};

class StateMachine {
public:
  StateMachine(State* initialState) : currentState {initialState} {}
  ~StateMachine() = default;

  bool doTransitionCallActions(const Transition& transition) {
    bool success = false;
    if (currentState == transition.getFromState() && transition.condition()) {
      if (currentState->exitAction != nullptr) {
        currentState->exitAction();
      }
      if (transition.action != nullptr) {
        transition.action();
      }
      currentState = transition.getToState();
      if (currentState->entryAction != nullptr) {
        currentState->entryAction();
      }
      success = true;
    }
    return success;
  }

private:
  State* currentState;
};

// file: light_fsm.h -> generated code

#include "fsm.h"

class LightFsm {
public:
  LightFsm() = default;
  ~LightFsm() = default;

  bool turnOff() {
   bool result = false;
   if (not result) {
     result = fsm.doTransitionCallActions(turnOff_on_off);
   }
   return result;
  }

  bool turnOn() {
   bool result = false;
   if (not result) {
     result = fsm.doTransitionCallActions(turnOn_off_on);
   }
   return result;
  }

  State offState {};
  State onState {};

  Transition turnOff_on_off {Transition(&onState, &offState)};
  Transition turnOn_off_on {Transition(&offState, &onState)};

private:
  StateMachine fsm {StateMachine(&offState)};
};

// file light_fsm.cpp -> test or application code

#include <iostream>
#include "light_fsm.h"

int main() {
  LightFsm lightFsm;

  // Inject actions
  lightFsm.offState.entryAction = [] {std::cout << "entering off state" << std::endl;};
  //lightFsm.offState.exitAction = [] {std::cout << "exiting off state" << std::endl;};

  //lightFsm.onState.entryAction = [] {std::cout << "entering on state" << std::endl;};
  lightFsm.onState.exitAction = [] {std::cout << "exiting on state" << std::endl;};

  lightFsm.turnOn_off_on.action = [] {std::cout << "turning on" << std::endl;};
  lightFsm.turnOff_on_off.action = [] {std::cout << "turning off" << std::endl;};

  // Simulate events
  lightFsm.turnOn();
  lightFsm.turnOff();
```

```
  std::cout << "off is off" << std::endl;
  lightFsm.turnOff();

  return 0;
}
```

Model

Model-Driven Development (MDD) requires a model as input for the code generator. Therefore, ChatGPT's next task is to reverse engineer a PlantUML state diagram that matches the above code. Here is both the model Source Code and its graphical representation:

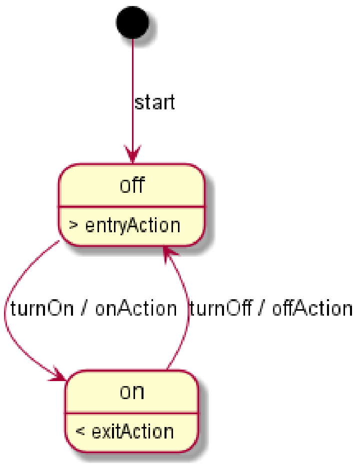
```
@startuml

[*] --> off : start
off --> on : turnOn / onAction
on --> off : turnOff / offAction

state off {
  off : > entryAction
}

state on {
  on : < exitAction
}

@enduml
```



Metamodel

The model is, of course, not enough; what is needed is a metamodel. This metamodel should contain the state and transition information found in the PlantUML models. A Python script is a practical approach to achieve this, and with an LLM available, it is more convenient to request the AI to generate the desired script rather than writing it manually. The following Source Code depicts the state of the script after some manual adjustments:

```python
#!/usr/bin/python3

import re
import argparse

def parse_plantuml(plantuml_code):
  state_pattern = re.compile(r'state (\w+)\s*{')
  entry_action_pattern = re.compile(r'\s*(\w+)\s*:\s*>\s*(.+)')
  exit_action_pattern = re.compile(r'\s*(\w+)\s*:\s*<\s*(.+)')
  transition_pattern = re.compile(r'(\w+)\s*-->\s*(\w+)\s*:\s*(\w+)\s*(\[.*\])?\s*/\s*(.+)')
  init_pattern = re.compile(r'\[\*\]\s*-->\s*(\w+)')

  states = []
  transitions = []

  lines = plantuml_code.split('\n')
  current_state = None

  for line in lines:
    state_match = state_pattern.match(line)
    if state_match:
      current_state = state_match.group(1)
      states.append({'name': current_state, 'entry_action': None, 'exit_action': None})

    entry_action_match = entry_action_pattern.match(line)
    if entry_action_match and current_state:
      states[-1]['entry_action'] = entry_action_match.group(2)

    exit_action_match = exit_action_pattern.match(line)
    if exit_action_match and current_state:
      states[-1]['exit_action'] = exit_action_match.group(2)

    transition_match = transition_pattern.match(line)
    if transition_match:
      from_state = transition_match.group(1)
      to_state = transition_match.group(2)
      event = transition_match.group(3)
      action = transition_match.group(4)
      transitions.append({'from_state': from_state, 'to_state': to_state, 'event': event, 'action': action})

    init_match = init_pattern.match(line)
    if init_match:
```

```
      init_state = init_match.group(1)

  return states, transitions, init_state

def events(transitions):
  events = set(transition['event'] for transition in transitions)
  return list(events)

def transitions_for_event(transitions, event):
  return [transition for transition in transitions if transition['event'] == event]

def indent(depth):
  return " " * depth * 3

def generate_cpp(name, states, transitions, init_state):
  pass # see Section 'Generator'

def generate_rust(name, states, transitions, init_state):
  pass # see Section 'Generator'

def main():
  parser = argparse.ArgumentParser(description="Parse PlantUML file and print collected information.")
  parser.add_argument("plantuml_file", help="Path to the PlantUML file")
  parser.add_argument("-l", "--language", choices=["c++", "rust"], help="Select between C++ and Rust")
  args = parser.parse_args()

  plantuml_file = args.plantuml_file

  try:
    with open(plantuml_file, 'r') as file:
      plantuml_code = file.read()
      name = plantuml_file.split('.')[0]
      states, transitions, init_state = parse_plantuml(plantuml_code)

      # Print collected information
      print("Name:", name)
      print("Initial State:", init_state)

      print("\nStates:")
      for state in states:
        print(state)

      print("\nTransitions:")
      for transition in transitions:
        print(transition)

      if args.language is not None:
        language = args.language.upper()
        if language == "C++":
          print("\nC++:\n")
          generate_cpp(name, states, transitions, init_state)
        elif language == "RUST":
          print("\nRust:\n")
          generate_rust(name, states, transitions, init_state)

  except FileNotFoundError:
    print(f"Error: File '{plantuml_file}' not found.")
    exit(1)

if __name__ == "__main__":
  main()
```

Generator

The following two subsections list the source code of the language specific generator functions generate_cpp() and generate_rust(), respectively.

Variant C++

The *fsm.h* header represents the static part of the simple FSM framework, while *light_fsm.cpp* serves as test or application code. In our example, the actual business logic generated from a model resides in *light_fsm.h*. The following function generate_cpp() extends the metamodel script to produce the desired *light_fsm.h* code (Source Code):

```
def generate_cpp(name, states, transitions, init_state):
  print("#include \"fsm.h\"\n")
  print("class " + name + " {")
  print("public:")
  print(indent(1) + name + "() = default;")
  print(indent(1) + "~" + name + "() = default;\n")
  transition_members = set()
  for e in events(transitions):
    print(indent(1) + "bool " + e + "() {")
    print(indent(2) + "bool result = false;")
```

```
    ets = transitions_for_event(transitions, e)
    for et in ets:
      print(indent(2) + "if (not result) {")
      transition_name = et['event'] + "_" + et["from_state"] + "_" + et["to_state"]
      transition_members.add((transition_name, et["from_state"], et["to_state"]))
      print(indent(3) + "result = fsm.doTransitionCallActions(" + transition_name + ");")
      print(indent(2) + "}")
    print(indent(2) + "return result;")
    print(indent(1) + "}\n")
  for s in states:
    print(indent(1) + "State " + s['name'] + "State {};")
  print()
  for t in transition_members:
    print(indent(1) + "Transition " + t[0] + " {Transition(&" + t[1] + "State, &" + t[2] + "State)};")
  print("\nprivate:")
  print(indent(1) + "StateMachine fsm {StateMachine(&" + init_state + "State)};")
  print("};")
```

Variant Rust

Now, with the language-independent model and a generator based on the metamodel available, addressing the portability quality attribute, especially when leveraging an LLM, seems obvious. Creating the Rust code with ChatGPT help was significantly more time-consuming compared to C++; it took quite a while to even compile. Well, I am sure this will get much better with more training data. However, I am less concerned with the results of the LLM and more concerned with my own superficial knowledge of this language. It is hard or even not possible to evaluate the results, push the LLM output in the right direction, or modify the code according to my own concepts if I do not have a complete oversight of all the language capabilities and specifics. This is not the right condition to produce quality. The idea of micro-modularity is not fulfilled, as I cannot assert that I clearly understand all possible side effects and consequences of the Rust code. My personal lessons learned: a deep dive into Rust is next on my agenda. Nevertheless, this is the example [Source Code]() after numerous editorial modifications:

```
// file: fsm.rs -> framework code

pub mod fsm {
  use std::collections::HashMap;
  use std::rc::Rc;

  pub struct State {
    pub entry_action: Option<Box<dyn Fn()>>,
    pub exit_action: Option<Box<dyn Fn()>>
  }

  impl State {
    fn new() -> Self {
      Self {
        entry_action: None,
        exit_action: None
      }
    }
  }

  #[derive(Clone)]
  pub struct Transition {
    from_state: String,
    to_state: String,
    pub condition: Rc<dyn Fn() -> bool>,
    pub action: Option<Rc<dyn Fn()>>
  }

  pub struct StateMachine {
    current_state: String,
    pub states: HashMap<String, State>,
    pub transitions: HashMap<String, Transition>
  }

  impl StateMachine {
    pub fn new(initial_state: &str) -> Self {
      let states = HashMap::new();
      let transitions = HashMap::new();
      //states.insert(String::from(initial_state), State::new());

      Self {
        current_state: String::from(initial_state),
        states,
        transitions
      }
    }

```

```
    pub fn add_state(&mut self, name: &str) {
      self.states.insert(String::from(name), State::new());
    }

    pub fn add_transition(&mut self, name: &str, from_state: &str, to_state: &str) {
      self.transitions.insert(String::from(name), Transition {
        from_state: String::from(from_state),
        to_state: String::from(to_state),
        condition: Rc::new(|| true),
        action: None
      });
    }

    pub fn do_transition_call_actions(&mut self, transition: Transition) -> bool {
      if self.current_state == transition.from_state && (transition.condition)() {
        if let Some(exit_action) = self.states.get(&self.current_state).unwrap().exit_action.as_ref() {
          exit_action();
        }
        if let Some(transition_action) = transition.action.as_ref() {
          transition_action();
        }
        self.current_state = transition.to_state.clone();
        if let Some(entry_action) = self.states.get(&self.current_state).unwrap().entry_action.as_ref() {
          entry_action();
        }
        return true;
      }
      false
    }
  }
}

// file light_fsm.rs -> generated code

pub mod light_fsm {
  use crate::fsm::fsm::StateMachine;

  pub struct LightFsm {
    pub fsm: StateMachine
  }

  impl LightFsm {
    pub fn new(initial_state: &str) -> Self {
     let mut fsm = StateMachine::new(initial_state);

     fsm.add_state("off");
     fsm.add_state("on");
     fsm.add_transition("turnOff_on_off", "on", "off");
     fsm.add_transition("turnOn_off_on", "off", "on");
     Self { fsm }
    }

    pub fn turn_off(&mut self) -> bool {
     let mut result = false;
     if !result {
       let transition = self.fsm.transitions["turnOff_on_off"].clone();
       result =self.fsm.do_transition_call_actions(transition)
     }
     result
    }

    pub fn turn_on(&mut self) -> bool {
     let mut result = false;
     if !result {
       let transition = self.fsm.transitions["turnOn_off_on"].clone();
       result =self.fsm.do_transition_call_actions(transition)
     }
     result
    }

  }
}

// file main.rs -> test or application code

mod fsm;
mod light_fsm;

use std::rc::Rc;
use crate::light_fsm::light_fsm::LightFsm;

fn main() {
  let mut light_fsm = LightFsm::new("off");

  // Inject actions
  light_fsm.fsm.states.get_mut("off").unwrap().entry_action = Some(Box::new(|| println!("entering off state")));
  //light_fsm.fsm.states.get_mut("off").unwrap().exit_action = Some(Box::new(|| println!("exiting off state")));

  //light_fsm.fsm.states.get_mut("on").unwrap().entry_action = Some(Box::new(|| println!("entering on state")));
```

```rust
    light_fsm.fsm.states.get_mut("on").unwrap().exit_action = Some(Box::new(|| println!("exiting on state")));

    light_fsm.fsm.transitions.get_mut("turnOn_off_on").unwrap().action = Some(Rc::new(|| println!("turning on")));
    light_fsm.fsm.transitions.get_mut("turnOff_on_off").unwrap().action = Some(Rc::new(|| println!("turning off")));

    // Simulate events
    light_fsm.turn_on();
    light_fsm.turn_off();
    println!("off is off");
    light_fsm.turn_off();
}
```

The following function generate_rust() extends the metamodel script to produce the desired *light_fsm.rs* code (Source Code):

```python
def change_case(str): # https://www.geeksforgeeks.org/python-program-to-convert-camel-case-string-to-snake-case/
  res = [str[0].lower()]
  for c in str[1:]:
    if c in ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'):
      res.append('_')
      res.append(c.lower())
    else:
      res.append(c)
  return ''.join(res)

def generate_rust(name, states, transitions, init_state):
  print("pub mod " + change_case(name) + " {")
  print(indent(1) + "use crate::fsm::fsm::StateMachine;\n")
  print(indent(1) + "pub struct " + name + " {")
  print(indent(2) + "pub fsm: StateMachine")
  print(indent(1) + "}\n")
  print(indent(1) + "impl " + name + " {")
  print(indent(2) + "pub fn new(initial_state: &str) -> Self {")
  print(indent(3) + "let mut fsm = StateMachine::new(initial_state);\n")
  for s in states:
    print(indent(3) + "fsm.add_state(\"" + s['name'] + "\");")
  for e in events(transitions):
    ets = transitions_for_event(transitions, e)
    for et in ets:
      transition_name = et['event'] + "_" + et["from_state"] + "_" + et["to_state"]
      print(indent(3) + "fsm.add_transition(\"" + transition_name + "\", \"" + et["from_state"] + "\", \"" + et["to_state"] + "\");")
  print(indent(3) + "Self { fsm }")
  print(indent(2) + "}\n")
  for e in events(transitions):
    ets = transitions_for_event(transitions, e)
    print(indent(2) + "pub fn " + change_case(e) + "(&mut self) -> bool {")
    print(indent(3) + "let mut result = false;")
    for et in ets:
      transition_name = et['event'] + "_" + et["from_state"] + "_" + et["to_state"]
      print(indent(3) + "if !result {")
      print(indent(4) + "let transition = self.fsm.transitions[\"" + transition_name + "\"].clone();")
      print(indent(4) + "result =self.fsm.do_transition_call_actions(transition)")
      print(indent(3) + "}")
    print(indent(3) + "result")
    print(indent(2) + "}\n")
  print(indent(1) + "}")
  print("}")
```

Conclusion

In this article, I propose an innovative (Creative, More Creative, Most Creative) approach to combine modern AI techniques with traditional concepts like MDD to advance the ongoing pursuit of high-quality code. Rather than tasking the LLM with generating extensive, cohesive pieces of code that may become less comprehensible to developers involved, the core idea is to break down larger parts into micro-modules – components so small that the corresponding code can be understood in minutes. These small fragments can then be optimized with the assistance of an LLM and assessed in isolation.

When addressing portability, it might be more effective to provide the LLM with a specification text as input instead of the original code. This approach helps avoid biasing the LLM with language-specific techniques and encourages a more idiomatic solution

Model-Driven Development (MDD) could serve as a method to systematically reintegrate generalized and optimized micro-modules back into the original larger context.

References

Source Code	*Source code of this article*, Nov, 2023. Available on GitHub.
Engel23a	Engel, Dirk, *Neither Human nor Machine*, May 2, 2023. Shared on LinkedIn and available on GitHub.
Engel23b	Engel, Dirk, *Creative, More Creative, Most Creative*, July 20, 2023. Shared on LinkedIn and available on GitHub.
Engel23c	Engel, Dirk, *Divide (along interfaces) and Conquer*, August 22, 2023. Shared on LinkedIn and available on GitHub.
Engel23d	Engel, Dirk, *A Software Dependency Flip Book*, October 31, 2023. Shared on LinkedIn and available on GitHub.