POS Projekt

Alessandro Crispino

April 2020

Contents

Chapter 1

Einleitung

Dieses Projekt wurde im Zuge der Aubesserung der POS-Note durchgeführt. Die Aufgabe war es einen Bankomaten, wie er in Österreich existiert, mittels der Library "[Boost].SML" zu implementieren. Dafür wurde zuerst ein UML-Zustandsdiagramm erstellt und dieses anschliessend mittels "[Boost].SML" umgesetzt.

Chapter 2

[Boost].SML

Figure 2.1: Logo [Boost].SML

Bei [Boost].SML handelt es sich handelt es sich um eine skalierbare header-only State Machine Library. Mittels dieser Bibliothek soll es ermöglicht werden unstrukturierten und unleserlichen Code zu verbessern. [Boost].SML ist ausserdem eine Verbesserung des Vorgängers Boost.MSM, welcher einige Nachteile hatte. Somit kann man [Boost].SML auch als Verbesserung von Boost.MSM definieren. [Boost].SML kann ab C++14 verwendet werden.

2.1 Installation

Um [Boost].SML verwenden zu können muss die Library zuerst installiert werden. Da es sich hierbei um eine header-only Library handelt, muss diese nur mit folgendem Kommando heruntergeladen werden:
wget https://raw.githubusercontent.com/boost-experimental/sml/master/include/boost/sml.hpp
Anschliessend muss die Header Date eingebunden und der “sml“ namespace

definiert werden. Dies funktioniert folgendermaßen:

```
#include "boost/sml.hpp"
```

```
namespace sml = boost::sml;
```

Wichtig ist noch zu erwähnen, dass [Boost].SML erst ab C++14 verwendet werden kann.

2.2 Features

Im folgenden Abschnitt werden die einzelnen Funktionen und Elemente der Bibliothek aufgezählt und erklärt.

2.2.1 States / Zustände und Events / Ereignis

Ein endlicher Automat besteht aus einer endlichen Anzahl von Zuständen und Übergängen. Diese Übergänge werden durch Ereignisse ausgelöst. Daher handelt es sich bei States und Events um Grundelemente der Library.

Implementierung States

States werden folgendermaßen implementiert:

```
auto Automat_Bereit_state = "Automat_Bereit"_s;
```

Ausserdem können Endzustände festgelegt werden welche folgend implementiert werden:

```
"Automat_Bereit"_s = X;
```

Zustände können zudem folgendermaßen ausgegeben werden:

```
std::cout << state.c_str() << std::endl;
```

Implementierung Events

Events können folgendermaßen implementiert werden:

```
struct Karte_Einfuehren_event{};
```

2.2.2 Guards und Actions

Bei Guards und Actions handelt es sich um aufrufbare Objekte welche vom Automaten ausgeführt werden um zu überprüfen ob ein Übergang stattfinden soll. Guards müssen hierbei einen booleschen Wert zurückliefern. Actions hingegen müssen keine Rückgabewert besitzen.

Implementierung Guards

Guards können wie folgt implementiert werden:

```
const auto geldentnahme = [](){
    //Code
    return true;
};
```

Implementierung Actions

Actions werden ähnlich wie Guards erstellt allerdings müssen diese, wie bereits erwähnt, nichts zurückgeben. Ausserdem können diese auch mittels Lambda Ausdrücken im Transition Table implementiert werden. Dies sieht folgendermaßen aus:

```
event<name\_event> / [] { cout << ``Karte wird eingef hrt`` < endl; }
```

Dazu später mehr

2.2.3 Transition Table

Um States, Events, Guards und Actions zu verbinden wird ein Transition Table benötigt. Dieser verbindet alle oben erwähnten Elemente und bildet daraus einen Automaten. Ein Transition Table kann folgendermaßen aussehen:

```
using namespace sml;

make_transition_table(
 *"src_state"_s + event<my_event> [ guard ] / action = "dst_state"_s
, "dst_state"_s + "other_event"_e = X
);
```

Ausserdem können bei der Implementierung eines solchen Tables zwei verschiedene Notationen verwendet werden, diese sind die Postfix und Prefix Notation.
Die folgende Auflistung zeigt die verschiedenen Funktion, in den verschiedenen Notationen:

Postfix:

- Übergang Zustand und Event mit Guard:
  state + event [guard]
- anonymer Übergang mit Action:
  src_state / [] = dst_state
- Selbstübergang:
  src_state / [] = src_state
- Übergang ohne Guard oder Action:
  src_state + event = dst_state

- Übergang mit Guard und Action:
  src_state + event [guard] / action = dst_state
- Übergang mit mehreren Guards und Actions:
  src_state + event [guard && (![]return true; && guard2)] / (action, action2, []) = dst_state

Prefix:

- Übergang Zustand und Event mit Guard:
  state + event [guard]
- anonymer Übergang mit Action:
  dst_state <= src_state / []
- Selbstübergang:
  src_state <= src_state / []
- Übergang ohne Guard oder Action:
  dst_state <= src_state + event
- Übergang mit Guard und Action:
  dst_state <= src_state + event [guard] / action
- Übergang mit mehreren Guards und Actions:
  dst_state <= src_state + event [guard && (![]return true; && guard2)] / (action, action2, [])

2.2.4 Startzustände

Da jeder Automat an einem bestimmten Punkt beginnen muss gibt es Startzustände. Diese definieren den Anfang eines Automaten. Startzustände werden mittels eines Sternchens "*" implementiert. Dies sieht folgendermaßen aus:

```
return make_transition_table(
    *start\_state + event<name\_event> = end\_state
);
```

2.2.5 State Machine

Um eine State Machine zu erstellen wird ein Transition Table benötigt. Dieser Transition Table wird mittels der Funktion "make_transition_table" in einem überladenen Operator "()" einer Klasse erstellt. Dies sieht folgendermaßen aus:

```
class test {
public:
  auto opeartor()() {
    using namespace sml;
    return make_transition_table(
```

```
      *"src_state"_s + event<my_event> [ guard ] / action = "dst_state"_s,
       "dst_state"_s + event<game_over> = X
     );
  }
};
```

Mittels folgendem Code wird anschließend eine State Machine erzeugt:

```
sml::sm<test> sm;
```

Ausserdem können den Guards und Actions Parameter mittels des Konstruktors übergeben werden. Um dies besser nachvollziehen zu können, dient folgendes Code-Beispiel:

```
auto guard = [](double d) { return true; }
                      |
                      \--------\
                               |
auto action = [](int i){}      |
                    |          |
                    |          |
                    \-\    /---/
                      |    |
sml::sm<example> s{42, 87.0};
```

2.2.6 Events ausführen

Um einen Automaten verwenden zu können, müssen die einzelnen Events ausgeführt werden. Ist dies im Transition Table richtig definiert so gelangt man von einem Zustand mittels der Ausführung eines Events in den nächsten. Dafür gibt es die Methode "process_event". Diese wird folgendermaßen verwendet:

```
sml::sm<example> sm;

sm.process_event(event{});
```

Events können außerdem auch im Transition Table ausgelöst werden. Das wird wie folgt gemacht:

```
start_state + event<my_event> / process(my_event_to_process{}) = end_state
```

2.2.7 Fehlerbehandlung

Falls es zu dem Fall kommen sollte, dass eine State Machine ein ausgelöstes Event nicht behandeln kann, kommt es zu einem "unexpected event". Ein solches kann auf folgende Weise abgefangen werden:

```
"start_state"_s + unexpected_event<some_event> = X
```

Ausserdem können auch unbekannte Events mit folgendem Code abgefangen werden:

```
"start_state"_s + unexpected_event<_> = X
```

2.2.8 Testen

Um den Automaten auch während des Erstellens testen zu können werden folgende Methoden bereitgestellt:

- state_machine.is: prüfen ob state_machine im angegebenen Zustand ist
- state_machine.set_current_states: setzt den derzeitigen Zustand der state_machine in den angegebenen Zustand
- state_machine.visit_current_states: gibt den derzeitigen Zustand der state_machine zurück

Chapter 3

Bankomat

Im folgenden Kapitel wird sich genauer mit der Implementierung des Bankomaten auseinandergesetzt. Hierfür wird zuerst das UML-Zustandsdiagramm gezeigt und anschließend die einzelnen Schritte der Implementierung erklärt.

3.1 UML-Zustandsdiagramm

Die Rechtecke mit den abgerundeten Ecken definieren die einzelnen Zustände, Übergänge werden mittels der Pfeile beschrieben. Ausserdem gibt es noch Start- und Endereignisse diese werden durch Kreise charakterisiert. Wobei ein Startereignis aus einem ausgefüllten Kreis besteht und ein Endereignis aus zwei ineinander liegenden Kreisen, wobei der Innere ausgefüllt wird.

3.2 Implementierung

3.2.1 Grundaufbau

Zuerst wurde ein neues Modul mit dem Namen “bankomat.cpp“ und der dazu gehörigen Header Datei angelegt. Der Bankomat wurde ausschließlich in diesem Modul erstellt. Ausserdem wurde ein eigener Namespace mit der Bezeichnung bankomat definiert.

3.2.2 States

Im nächsten Schritt wurden alle Zustände des Zustandsdiagramms definiert. Dies wurde auf folgende Weise gemacht:

```
auto machine_ready_state = "Automat_Bereit"_s;
auto pin_request_state = "PIN_Aufforderung"_s;
auto menue_state = "Menue"_s;
auto bank_balance_state = "Kontostand"_s;
auto money_selection_state = "Geldauswahl"_s;
```

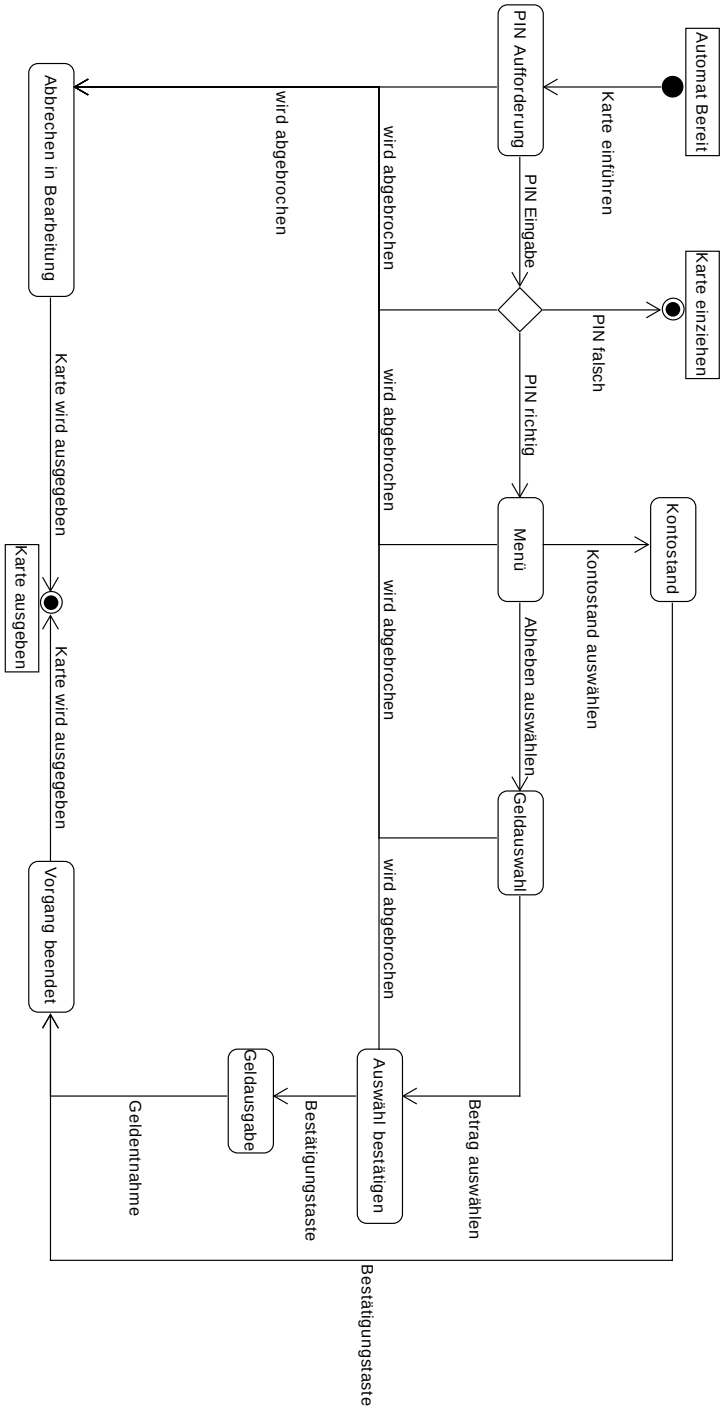

Figure 3.1: Zustandsdiagramm Bankomat

```
auto confirm_selection_state = "Auswahl_Bestaetigen"_s;
auto money_output_state = "Geldausgabe"_s;
auto withdraw_card_state = "Karte_Einziehen"_s;
auto cancel_in_progress_state = "Abbrechen_In_Bearbeitung"_s;
auto operation_ended_state = "Vorgang_Beendet"_s;
```

3.2.3 Events

Nachdem alle States definiert worden sind, sind anschließend alle Events definiert worden. Dies wurde folgendermaßen gemacht:

```
struct input_card_event{};
struct input_pin_event{};
struct select_balance_event{};
struct select_withdraw_event{};
struct confirm_event{};
struct confirm_amount_event{};
struct withdraw_money_event{};
struct cancel_event{};
struct card_output_event{};
```

3.2.4 Transition Table

Um das ganze letztendlich verbinden zu können wurde eine Transition Table angelegt welcher die Logik des Automaten implementierte. Zudem wurden Guards und Actions angelegt. Dies wurde auf folgende Weise implementiert:

```
return make_transition_table(
    *machine_ready_state +
        event<input_card_event>
            / [] { cout << "Karte wird eingefuehrt" << endl; }
                = pin_request_state,

    pin_request_state +
        event<input_pin_event>[right_PIN]
            / [] { cout << "PIN ist richtig" << endl; }
                = menue_state,

    pin_request_state +
        event<input_pin_event>[!right_PIN]
            / [] { cout << "PIN ist falsch, Karte wird eingezogen" << endl; }
                = withdraw_card_state,

    pin_request_state +
        event<cancel_event>
        = cancel_in_progress_state,
```

```
menue_state +
    event<select_balance_event>
        / [] { cout << "Option Kontostand wurde gewaehlt" << endl; }
            = bank_balance_state,

menue_state
    + event<select_withdraw_event>
        / [] { cout << "Option Abheben wurde gewaehlt" << endl; }
            = money_selection_state,
menue_state
    + event<cancel_event>
        / [] { cout << "Vorgang wird abgebrochen" << endl; }
            = cancel_in_progress_state,

bank_balance_state
    + event<confirm_event>
        / [] { cout << "Option wurde bestaetigt" << endl; }
            = operation_ended_state,

money_selection_state
    + event<confirm_amount_event>
        = confirm_selection_state,

money_selection_state
    + event<cancel_event>
        / [] { cout << "Vorgang wird abgebrochen" << endl; }
            = cancel_in_progress_state,

confirm_selection_state
    + event<confirm_event>
        / [] { cout << "Auswahl bestaetigen" << endl; }
            = money_output_state,

confirm_selection_state
    + event<cancel_event>
        / [] { cout << "Vorgang wird abgebrochen" << endl; }
            = cancel_in_progress_state,

money_output_state
    + event<withdraw_money_event>[withdraw]
        / process(card_output_event{})
            = operation_ended_state,

operation_ended_state
    + event<card_output_event>
```

```
            / [] { cout << "Karte wird ausgegeben" << endl; }
                = X,

    cancel_in_progress_state
        + event<card_output_event>
            / [] { cout << "Karte wird ausgegeben" << endl; }
                = X
);
```

Hier wurde jeweil ein Ausganszustand mit einem Event verbunden. Dies resultierte anschließend in einem weiteren Zustand. Guards wurden mittels eckiger Klammern aufgerufen. Actions wurden mittels Lambda Aufrufen realisiert, dafür wurde lediglich nach dem "/" ein Lambda Ausdruck definiert. Ausserdem wurde der Startzustand mittels eines Sternchens am Anfang, und ein Endzustand mittels eines "X" definiert.

3.2.5 PIN

Um es zu ermöglichen den Pin zu überprüfen wurde eine Struktur PIN erstellt. Diese wurde lediglich verwendet um einen Wert und zwar die richtige Zahlenkombination, zu speichern. Die Struktur war wie folgt aufgebaut:

```
struct PIN{
    std::string value{};
};
```

3.2.6 Guards

Guards werden verwendet um Übergänge nur dann zu ermöglich, wenn diese mittels Guards "verifiziert" werden. In diesem Projekt wurden hierfür zwei Guards erstellt, diese sind:

withdraw

Dieser Guard diente zur Bestätigung, welche mittels Betätigung der Enter Taste realisiert wurde. Dies wurde mittels der Methode "cin.get()" erzielt. Der Aufbau dieses Guards war folgender:

```
const auto withdraw = []() {
    cout << "Bitte das Geld entnehmen."
         << "Um das Geld zu entnehmen Enter Taste druecken."
         << endl;
    cin.clear();
    cin.ignore();
    cin.get();
    cout << "Geld entnommen." << endl;
    return true;
};
```

right_PIN

Im Guard "right_PIN" wurde die Eingabe des Pins überprüft. Da es eine fixe Pin gab, wurde dies einfach auf folgende Weise überprüft:

```
const auto right_PIN = [](PIN &pin) {
    return !pin.value.compare("1234");
};
```

Um hierbei auf die Dependencies der State Machine zugreifen zu können, wurde der Guard in der Struktur des Bankomaten implementiert.

3.2.7 Methode start()

In der Funktion "start()" wurde die gesamte Logik des Automaten abgewickelt. Hierfür wurde anfangs Objekt der Struktur Pin angelegt, welche anschließend bei der Initialisierung der State Machine als Parameter übergeben wurde. Außerdem wurde der State Machine noch eine "process_queue" übergeben welche dazu diente, dass events auch in Actions ausgelöst werden können.

Warten auf Bankomatkarte

Zu Beginn wurde mittels der Methode "cin.ignore()" auf das Einführen der Karte gewartet, ist diese mittels drücken der Enter Taste beendet worden, wurde anschliessend das Event "input_card_event()" ausgelöst.

PIN Eingabe

Als nächstes wurde der Pin Code des Users abgefragt. Hierbei wurde wurde bei der Eingabe überprüft, dass nur Zahlen eingegeben worden sind. Die Überprüfung des Pins auf Richtigkeit wurde anschließend im unter Punkt 3.2.6 erwähnten Guards gemacht. War die Eingabe des Pins erfolgreich gelang man weiter ins Menü. Andernfalls wurde die Karte eingezogen und der Bankomat beendet.

Menü

Im Menü kann zwischen zwei Optionen gewählt werden, diese sind "Kontostand" und "Geld abheben". Die Auswahl wurde mittels eingeben der Zahlen 1 und 2 getroffen, wobei 1 für "Kontostand" und 2 für "Geld abheben" verwendet wurde.

Option Kontostand

In der Option "Kontostand" wurde ein fixer Kontostand ausgegeben und anschließend auf die Bestätigung mittels drücken der Enter Taste gewartet. Nach der Bestätigung wurde der Bankomat beendet.

Option Geld abheben

In der Option “Geld abheben“ wird man zuerst aufgefordert einen Wunschbetrag einzugeben, welchen man beabsichtigt abzuhben. Dieser Betrag darf nur aus Zahlen bestehen und muss positiv sein. Ist eine korrekter Betrag eingegeben worden, wird man letztendlich aufgefordert seine Auswahl mittels eingeben der Zahl 1 zu bestätigen. Wurde die Auswahl bestätigt wird der Benutzer aufgefordert sein Geld zu entnehmen, dies wurde ebenfalls mittels drücken der Enter Taste realisiert. Zum Schluss wurde die Karte ausgegeben und das Programm hat sich beendet.

Abbrechen

Außerdem wurde es dem Benutzer in ausgewählten, im Zustandsdiagramm Abb. 3.1 ersichtlich, Zuständen den Vorgang abzubrechen. Dies wurde mittels eingeben der Taste “X“ oder “x“ umgesetzt. Wurde der Vorgang abgebrochen, wird die Karte ausgegeben und das Programm beendet.