Bankomat mittels Boost.[SML]

von Jakob Wildt

3. Mai 2020

1 Eidesstaatliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen und Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

2 Einleitung

Die folgende Dokumentation beschreibt das Projekt "Bankomat [Boost].SML", ein Schulprojekt, das durchgeführt wurde, um eine Note zu verbessern.
Es handelt sich um einen Automaten, der über die Kommandozeile bedient werden kann und auf Basis eines österreichischen Bankomaten implementiert wurde.

3 [Boost].SML

[Boost].SML, die "Weiterentwicklung"von Boost.MSM ist eine header-only State Machine Library. Die Bibliothek dient zur Vereinfachung und Strukturierung von kompliziertem State Machine-Code in C++ und kann ab dem C-Standard 2014 verwendet werden.

3.1 Installation

Die Library kann unter dem Link: https://boost-experimental.github.io/sml/index.html heruntergeladen werden. Daraufhin muss sie nur mehr mithilfe eines include-Befehls eingebunden werden.

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

3.2 Funktionen

3.2.1 States

Die Zustände eines Automaten werden durch States beschrieben. In [Boost].SML müssen die Zustände nicht unbedingt im Vorhinein definiert werden. Eine Definition im Transition Table reicht aus, dazu später mehr.

3.2.2 Events

Events führen Zustandsübergänge herbei und müssen im Gegensatz zu den Zuständen definiert werden, bevor sie im Transition Table verwendet werden können. Sie können als Instanz angelegt werden,

```
auto event = sml::event<my_event>;
```

oder als neuer Datentyp implementiert werden.

```
struct my_event { ... };
```

3.2.3 Guards und Actions

Guards und Actions sind Objekte, die von der State Machine verarbeitet werden, um zu überprüfen, ob ein Zustandsübergang (manchmal gefolgt von einer Aktion) durchgeführt werden soll. Guards returnen immer einen boolschen Wert, während Actions nicht zwingend etwas zurückgeben müssen.

Beispiele für die Implementierung von Guards wären:

```
auto guard1 = [] {
  return true;
};

auto guard2 = [](int, double) { // guard with dependencies
  return true;
};

const auto right_PIN = [](PIN& pin, Karte& karte){
    std::cout << "PIN VALUE: " << pin.value << std::endl;
    return pin.value == karte.pin;
};
```

Actions wurden in diesem Projekt nicht verwendet, hier sind jedoch die Beispiele des offiziellen [Boost].SML Tutorials:

```
auto action1 = [] { };
auto action2 = [](int, double) { }; // action with dependencies
auto action3 = [](int, auto event, double) { }; // action with an event d
struct action4 {
    void operator()() noexcept { }
};
```

3.2.4 State Machine und Transition Table

Um eine State Machine zu erstellen, braucht man in [Boost].SML einen Transition Table - also eine Zustandsübergangstabelle. Sie beschreibt in welchem Zustand welcher Input welchen Zustandsübergang bewirkt. Der Automat wird als eigene Klasse definiert und eine Instanz davon angelegt um die Funktionen auszuführen.

```
namespace sml = boost::sml;

namespace {
struct e1 {};
struct e2 {};
struct e3 {};

struct transitions {
  auto operator()() const noexcept {
    using namespace sml;
    return make_transition_table(
       *"idle"_s                    / []
       { std::cout << "anonymous transition" << std::endl; } = "s1"_s
      , "s1"_s + event<e1>          / []
      { std::cout << "internal transition" << std::endl; }
      , "s1"_s + event<e2>          / []
      { std::cout << "self transition" << std::endl; } = "s1"_s
      , "s1"_s + sml::on_entry<_> / []
      { std::cout << "s1 entry" << std::endl; }
      , "s1"_s + sml::on_exit<_>  / []
      { std::cout << "s1 exit" << std::endl; }
      , "s1"_s + event<e3>          / []
      { std::cout << "external transition" << std::endl; } = X
    );
  }
};
}
```

```
int main() {
  sml::sm<transitions> sm;
  sm.process_event(e1{});
  sm.process_event(e2{});
  sm.process_event(e3{});
  assert(sm.is(sml::X));
}
```

4 Bankomat

Im folgenden Abschnitt wird die Funktionsweise, sowie die Bedienung des Bankomaten genauer erklärt.

4.1 UML-Diagramm

Das UML-Diagramm soll den Ablauf veranschaulichen und alle möglichen Zustandsänderungen anzeigen. Die Rechtecke beschreiben die Zustände und die Kanten die Zustandsübergänge. Zustandsübergänge sind oft mit aktion1/aktion2 beschriftet, wobei "aktion1" für die Eingabe steht, die den Zustandsübergang zur Folge hat und "aktion2" für die mögliche Ausgabe, die aus dem Zustandswechsel folgt.

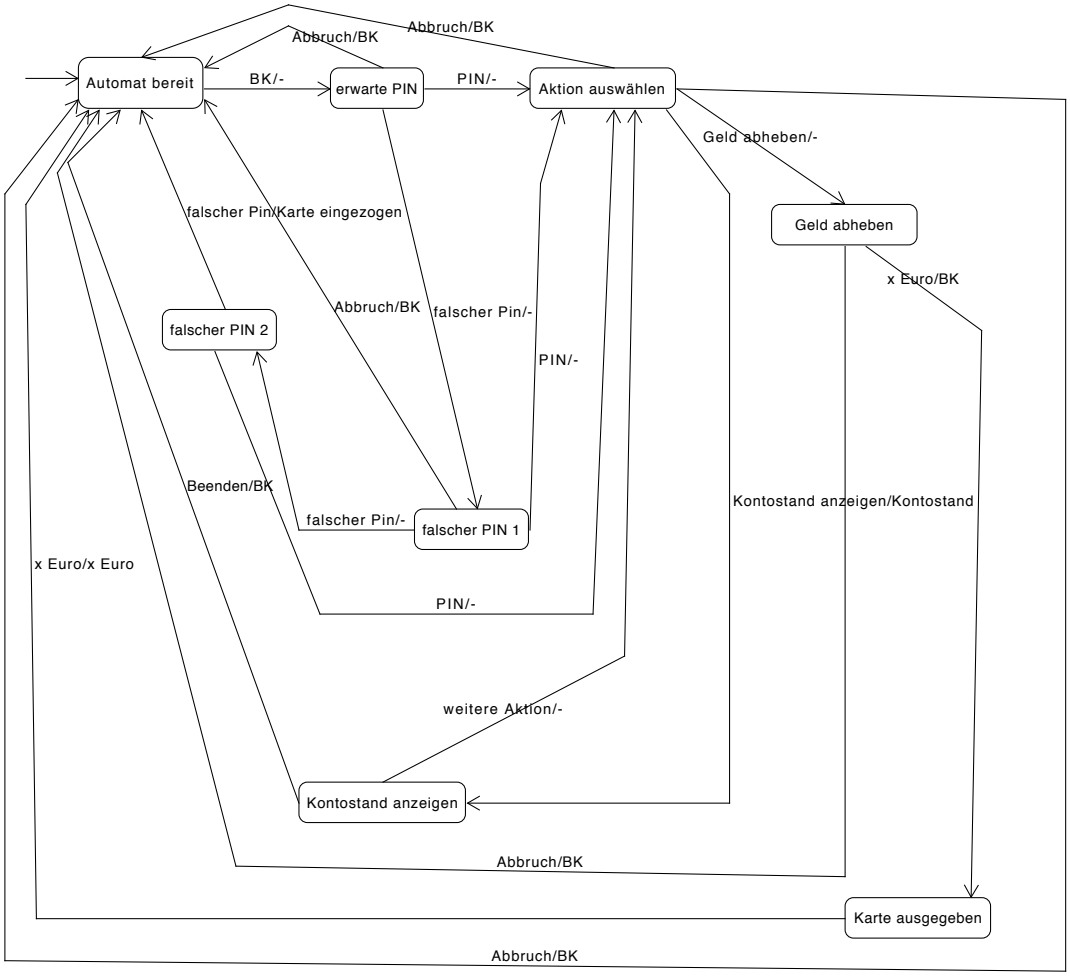

Abbildung 1: Das UML-Zustandsdiagramm des Bankomaten.

4.2 Code

4.2.1 Aufbau

Der Automat wurde in einem header-only Modul Bankomat entwickelt. In der main-File wird eine Instanz des Automaten angelegt und per Kommando

```
sm.process_event()
```

werden die Zustandsübergänge herbeigeführt.

4.2.2 Zustände

Wie man aus dem UML herauslesen kann, gibt es 8 verschiedene States, in denen sich der Automat befinden kann. Die Zustände werden erst im Transition Table definiert und verwendet. Siehe Codebeispiel Zustandsübergänge.

4.2.3 Events

Die Events führen immer zu einem Zustandsübergang, der manuell durch die Funktion `process_event()` ausgelöst werden kann.

```
struct karte_eingef{
};
struct pin{
};
struct abbruch{
};
struct geld_abheben_e{
};
struct kontostand_e{
};
struct x_euro{
};
struct weitere_aktion{
};
```

Sie werden einfach durch leere Strukturen definiert und in weiterer Folge von der State Machine verarbeitet.

4.2.4 Zustandsübergänge - Transition Table

Der Transition Table definiert aus welchem Ausgangszustand die State Machine durch welches Event in welchen Folgezustand wechselt. Hier wird auch der Startzustand durch einen * vor dem Namen definiert und die Guards und Actions werden bei den Zustandsübergängen aufgerufen.

```
struct Automaton {

public:
  auto operator()() {
    return make_transition_table(

     *"automat_bereit"_s + event<karte_eingef> / []
     {} = "erwarte_pin"_s,

      "automat_bereit"_s + on_entry<_> / [] {
          std::cout << "Karte bitte!" << std::endl;
          std::cin >> karte;

          bool check_card{true};

          while(check_card){
                if(cards.find(karte) == cards.end()){
                std::cout << "Unguelitge Karte. Karte bitte!"
                << std::endl;
                std::cin >> karte;
            }
            else{
                karte_pin.pin = cards.at(karte);
                check_card=false;
            }
          }
      },
```

```
        "erwarte_pin"_s + event<pin> [right_PIN] / []
        {} = "aktion_auswahlen"_s,

        "erwarte_pin"_s + event<abbruch> / []
        {} = "automat_bereit"_s,

        "erwarte_pin"_s + event<pin> [!right_PIN] / []
        {} = "falscher_pin1"_s,

        "erwarte_pin"_s + on_entry<_> / []
        {
            std::cout << "PIN eingeben!" << std::endl;
            std::cin >> pin_;
            pin_inp.value = pin_;
        },
        "falscher_pin1"_s + event<pin> [!right_PIN] / []
        {} = "falscher_pin2"_s,

        "falscher_pin1"_s + event<pin> [right_PIN] / []
        {} = "aktion_auswahlen"_s,

        "falscher_pin1"_s + on_entry<_> / [] {
            std::cout << "Fehler 1! PIN erneut eingeben!" << std::endl;
            std::cin >> pin_;
            pin_inp.value = pin_;
        },
        "falscher_pin2"_s + event<pin> [!right_PIN] / [] {
            std::cout << std::endl << "Karte eingezogen!" << std::endl;
            cards.erase(karte);
            } = "automat_bereit"_s,
    );
  }
};
```

Hier ein Ausschnitt aus dem Transition Table dieses Projektes. Der Echte ist noch um einiges länger, dieser Teil soll nur zeigen, wie man sich diese Zustandsübergangstabelle vorstellen kann. Einige dieser Zeilen sind keine Definitionen für Übergänge, sondern beschreiben das Verhalten der State Machine beim Eintritt in verschiedene Zustände. Oft wird eine Eingabe erwartet und ein kurzer Text zur Orientierung ausgegeben.

Hier ein Beispiel für einen Zustandsübergang:

```
*"automat_bereit"_s + event<karte_eingef> / []{} = "erwarte_pin"_s;
```

.

Und hier ein Beispiel für eine on_entry-Aktion, die durchgeführt wird, sobald der Automat in diesen Zustand eintritt:

```
"automat_bereit"_s + on_entry<_> / [] {
            std::cout << "Karte bitte!" << std::endl;
            std::cin >> karte;

            bool check_card{true};

            while(check_card){
                  if(cards.find(karte) == cards.end()){
                  std::cout << "Unguelitge Karte. Karte bitte!"
                  << std::endl;
                  std::cin >> karte;
              }
              else{
                  karte_pin.pin = cards.at(karte);
                  check_card=false;
              }
            }
        };
```

4.2.5 Guards

Guards sollen Zustandsübergänge mit falschen oder ungültigen Werten verhindern. Nur, wenn der guard `true` zurückliefert geht der Zustandsübergang durch.

Definition:

```
const auto right_PIN = [](PIN& pin, Karte& karte){
    std::cout << "PIN_VALUE:" << pin.value << std::endl;
    return pin.value == karte.pin;
};
```

Anwendung:

```
"erwarte_pin"_s + event<pin> [right_PIN] / [] {} = "aktion_auswahlen"_s;
```

Dieser Guard vergleicht den eingegebenen (pin.value) mit dem richtigen, in der Liste eingatragenen PIN (karte.pin). Je nachdem, ob die beiden übereinstimmen, wird der Zustandsübergang zugelassen oder nicht.

4.2.6 Login

Um sich in den Bankomat einzuloggen, muss man zuerst seine Karte ”einlesen”. Das bedeutet, einfach entweder ”karte1”, ”karte2” oder ”karte3” einzugeben, wenn man nach der Karte gefragt wird. Hat man diesen Schritt erfolgreich erledigt, wird man nach dem PIN gefragt. Der PIN für karte1 ist 1234, für karte2 6969 und für karte3 0420.

4.2.7 Aktion auswählen

In diesem Zustand wird der Benutzer gefragt, was er als nächstes machen will. Seine Möglichkeiten bestehen daraus, sich den Kontostand anzeigen zu lassen, Geld abzuheben oder den Vorgang abzubrechen und sich auszuloggen. Sollte er letzteres wählen, begibt sich der Automat wieder in den Startzustand und wartet auf den Input einer Karte.

4.2.8 Kontostand anzeigen

Wählt der Benutzer die 2. Option Kontostand anzeigen aus, wird ihm der aktuelle Kontostand der Karte angezeigt, mit der er eingeloggt ist, bevor er automatisch

wieder in den Zustand ”Aktion auswählen” geleitet wird.

4.2.9 Geld abheben

Bei der ersten Auswahlmöglichkeit wird der User gefragt, wie viel Geld er abheben will. Hier ist es ihm nicht möglich, mehr als 400 Euro, oder ins Minus hinein abzuheben. Sobald die Transaktion abgeschlossen ist und das Geld vom Konto abgezogen wurde, wird der Benutzer automatisch ausgeloggt und der Automat befindet sich wieder im Startzustand und wartet auf eine Eingabe.