Boost your State-machines

Create State-machines that are easy to read and maintain

Paweł Wiśniewski – consulting@wisniewski.io

State-machines sind ein wichtiger Bestandsteil von Software. Leider sind sie zu oft per Hand implementiert z.B. unter Verwendung von if-else oder switch-case Konstruktionen. Solche Konstruktionen sind schwer zu verstehen und schwierig zu erweitern. Zusätzlich sehen sie immer ein bisschen anders aus, weil sie stets wieder von vorne geschrieben werden.
Der Vortrag zeigt eine Lösung auf Basis der Bibliothek Boost.SML. Die mit der Bibliothek erstellten State-machines sind nicht nur einfacher zu verstehen und zu erweitern, sie sind auch oft effizienter als die handgeschrieben Varianten.

Was sind State-machines (Zustandsautomaten) und wofür sind sie gut?
Embedded Systeme sind meistens Event getrieben, das heißt sie warten auf verschiedene (interne oder externe) Ereignisse. Wenn ein Ereignis auftritt, wird es abgearbeitet. Innerhalb der Verarbeitung können weitere interne Ereignisse generiert werden. Wenn es kein Ereignis zum Verarbeiten gibt, wird wieder gewartet.
Dieses Verhalten lässt sich als sequentielle Software realisieren. Wenn die Reihenfolge von Events nicht bekannt ist oder wenn auf mehrere Events gewartet wird, wird eine sequentielle Software komplex. Es ist deutlich einfacher, ein event-getriebenes Verhalten mit Zustandsautomaten abzubilden.

Zustandsautomaten lassen sich mit Hilfe von UML- Zustandsdiagrammen[1] graphisch projektieren. Ein Zustandsdiagramm zeigt die zur Laufzeit erlaubten Zustände eines Zustandsautomaten an und gibt Ereignisse an, die seine Zustandsübergänge auslösen.

Die Zustände in einem Zustandsdiagramm werden durch Rechtecke mit abgerundeten Ecken dargestellt. Die Pfeile zwischen den Zuständen symbolisieren mögliche Zustandsübergänge. Sie sind mit den Ereignissen beschriftet, die zu dem jeweiligen Zustandsübergang führen.

Im Diagramm 1 sind die wichtigsten UML Elemente zu sehen.

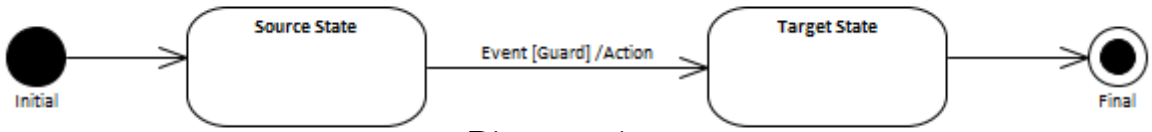

Diagramm 1

Initial - Startpunkt des Zustandsautomaten
State – ein Zustand
Transition - Zustandsübergang
Event – Ereignis, das einen Zustandsübergang auslöst
Guard - Bedingung für die Transition
Action - Aktion während der Transition
Final - Ende des Zustandsautomaten

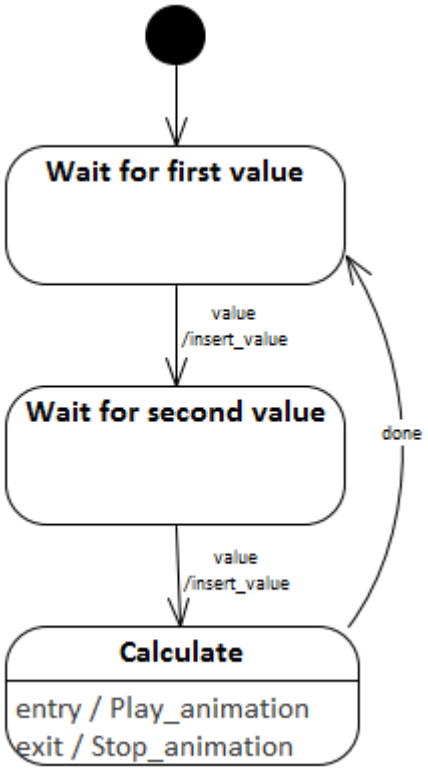

Diagramm 2

```
01: void handle(auto event) {
02:   switch(state)
03:   {
04:     case WaitForFirstEvent:
05:      if(event == value) {
06:        insert_value()
07:        state = WaitForFirstEvent
08:      }
09:      break;
10:     case WaitForSecondEvent:
11:      if(event == value) {
12:        insert_value()
13:        state = Calculate;
14:        Play_animation();
15:      }
16:      break;
17:     case Calculation()
18:      if(event == done) {
19:        state = WaitForFirstEvent
20:        Stop_animation();
21:      }
22:      break;
23:   }
24: }
```

Listing 1

Handgeschriebene Implementierungen (switch-case, if-else)

Betrachten wir den Zustandsautomaten aus Diagramm 2. Er beinhaltet drei States (Wait for first value, Wait for second value und Calculate), zwei Events (value und done) und eine Aktion (insert_value).

Es wäre möglich, diese state-machine mit Hilfe von if-else oder switch-case Konstruktionen zu implementieren. Der Quellcode könnte wie in Listing 1 (Switch-case) oder Listing 2 (if-else) aussehen. Leider ist auf den ersten Blick in beiden Fällen schwierig zu erkennen, was diese Software macht. Die Entry Aktion von State Calculate (Play_animation) ist in beiden Fällen (if-else und switch-case) mehr eine Exit Aktion von State WaitForSecondValue. Das kann zu Schwierigkeiten führen, wenn das System um weitere Transitionen in Richtung Calculate erweitert werden soll.

```
01: void handle(auto event) {
02:  if (event == value && !isAnimating) {
03:    insert_value();
04:    ++valuesCount;
05:
06:    if (valuesCount == 2) {
07:      Play_animation();
08:      isAnimating = true;
09:    }
10:  } else if(event == done && isAnimating) {
11:    Stop_animation();
12:    isAnimating = false;
13:    valuesCount= 0;
14:  }
15: }
```

Listing 2

Wenn es mehrere Events und Transitionen im System gibt, wird der Quellcode lang und unübersichtlich. Wenn wir unsere Software um weitere States erweitern möchten, müssen wir alle Bedingungen betrachten - auch diejenigen, die nicht leicht erkennbar sind.

Wir versuchen den Zustandsautomaten von Diagramm 2 um ein einfaches Error-Handling zu erweitern, wie in Diagramm 3 zu sehen ist.

Der Quellcode (Listing 3) ist nun fast doppelt so lang; eine extra Ebene mit einer if-Bedingung ist dazugekommen. Jetzt haben wir auch an zwei Stellen

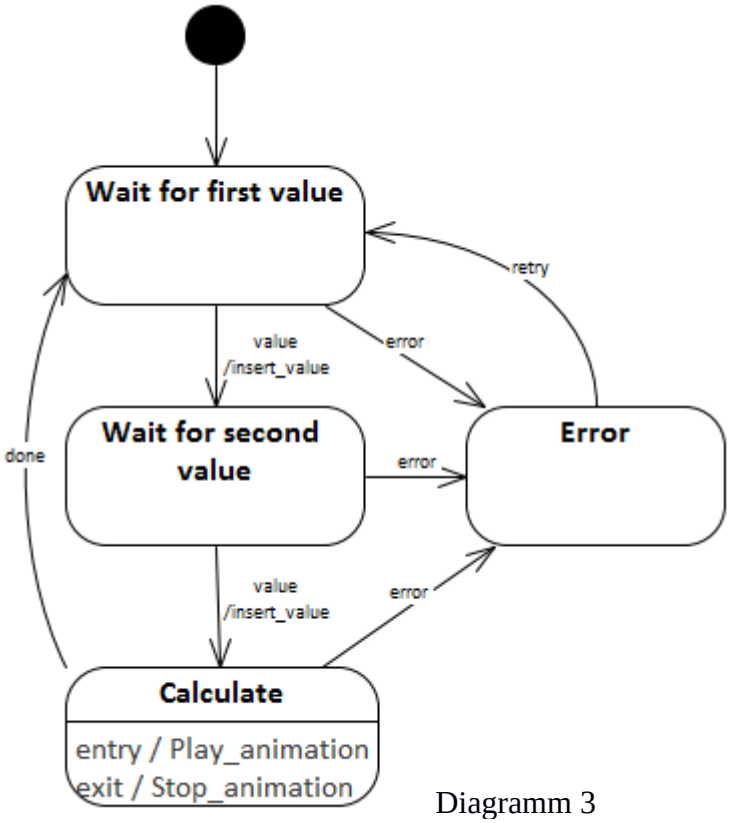

Diagramm 3

```
01: void handle(auto event) {
02:   if(errorHandling)
03:   {
04:     if(event == retry) {
05:       errorHandling = false;
06:     }
07:   } else {
08:     if(event == error) {
09:       errorHandling = true;
10:       if(isAnimating) {
11:         Stop_animation()
12:         isAnimating = false;
13:         valuesCount = 0;
14:       }
15:     } else if (event == value && !isAnimating) {
16:      insert_value();
17:      ++valuesCount;
18:
19:      if (valuesCount == 2) {
20:         Play_animation();
21:         isAnimating = true;
22:       }
23:     } else if(event == done && isAnimating) {
24:      Stop_animation();
25:      isAnimating = false;
26:      valuesCount = 0;
27:     }
28:   }
29: }
```

Listing 3

Stop_animation (in Zeile 11 und 24) als Exit Aktionen für Calculate State. Bei realistischen Zustandsautomaten ergeben sich noch deutlich mehr Duplikate.

Die Grundlagen der Bibliothek [Boost].SML[3]
Das Verhalten von Zustandsdiagramm 2 lässt sich mit Hilfe von UML 2.5 („Textuelle Repräsentation“) als Zustandsautomat-Transitionen-Tabelle definieren (Listing 4).

```
transition_table {
 *Wait_For_First_Value + value / insert_value ->
Wait_For_Second_Value,
  Wait_For_Second_Value + value / insert_value -> Calculate,
  Calculate + on_entry / play_animation(),
  Calculate + on_exit / stop_animation(),
  Calculate + done -> Wait_For_First_Value
};
```

Listing 4

In der Transitionen-Tabelle können wir einfach sehen, unter welchen Bedingungen wir von State zu State kommen. Wenn wir unsere Transitionen-Tabelle um ein Error Handling erweitern wollen, müssen wir nur die neuen Transitionen eintragen (Listing 5).
Das Herz von [Boost].SML ist die Transitionen-Tabelle. Die Syntax basiert auf der gezeigten UML-Notation:

```
SourceState + event [Guard] / Action = TargetState
```

Die Transitionen-Tabelle von Listing 4 kann man für die Nutzung mit dem [Boost].SML Framework fast unverändert übernehmen (Listing 6).

```
transition_table {
 *Wait_For_First_Value + value / insert_value ->
Wait_For_Second_Value,
  Wait_For_First_Value + error -> Error,
  Wait_For_Second_Value + value / insert_value -> Calculate,
  Wait_For_Second_Value + error -> Error,
  Calculate + on_entry / play_animation(),
  Calculate + on_exit / stop_animation(),
  Calculate + done -> Wait_For_First_Value,
  Calculate + error -> Error
};
```

Listing 5

```
return make_transition_table(
 *"Wait_For_First_Value"_s + value / insert_value =
"Wait_For_Second_Value"_s,
   "Wait_For_Second_Value"_s + value / insert_value =
"Calculate"_s,
   "Calculate"_s + on_entry / play_animation,
   "Calculate"_s + on_exit / stop_animation,
   "Calculate"_s + done = "Wait_For_First_Item"_s
};
```

Listing 6

Die Erweiterung der Tabelle ist genauso einfach wie in Listing 5.

Außer der Transitionen-Tabelle, müssen wir alle Events, Aktionen und Guards definieren. Die Aktionen und Guards sind Callable [2] Elemente, wie z.B. Lambda-Funktionen (Listing 7). Der Unterschied zwischen den Aktionen und Guards ist, dass die Aktionen nichts zurückgeben dürfen und die Guards bool zurückgeben müssen. Events sind benutzerdefinierte Typen wie z.B. Strukturen (Listing 8).

```
auto insert_value = [](auto e) { do_something_1() };
auto play_animation = [](auto e) { do_something_2() };
auto stop_animation = [](auto e) { do_something_3() };
```

Listing 7

```
struct value {};
struct done {};
```

Listing 8

Der fertige Zustandsautomat aus Diagramm 2 ist in Listing 9 zu sehen. Dort sind gleich alle Zustände (Zeile 6-8), Aktionen (Zeile 11-13), Events (Zeile 2-3) und die Transitionen-Tabelle (Zeile 19-25) zu erkennen. Der gesamte Quellcode ist gut lesbar und lässt sich einfach mit Diagramm 2 vergleichen. Änderungen sind leicht durchzuführen, weil sich die Logik des Zustandsautomaten an einer Stelle befindet und einfach zu verstehen ist.

In Zeile 29 bis 44 ist eine einfache Anwendung eines Boost::SML Zustandsautomaten gezeigt. Es ist lediglich notwendig, den Zustandsautomaten zu instantiieren (Zeile 32) und dann können wir mit Hilfe von process_event Transitionen in unserem Zustandsautomaten auslösen (Zeilen 36, 39 und 42).

Zusätzlich sehen wir in Zeile 33 die Prüfung des RAM-Verbrauchs, durch die Instanz des Zustandsautomaten. Wenn die Instanz größer als ein Byte ist, dann bekommen wir einen Fehler zur Compilezeit. Weitere Prüfungen sind in Zeilen 34, 40 und 43 zu sehen. Dort wird geprüft, ob die Transitionen richtig definiert sind. Unser Zustandsautomat soll im Zustand Wait_For_First_Value starten, diese Prüfung findet in Zeile 34 statt. Weitere Prüfungen werden nach Aufruf von process_event durchgeführt. Diese Prüfungen finden zur Laufzeit statt und können in Unit-Tests verwendet werden.

```
01: //events:
02: struct value {};
03: struct done {};
04:
05: //states:
06: const auto Wait_For_First_Value = state<class
Wait_For_First_Value>;
07: const auto Wait_For_Second_Value = state<class
Wait_For_Second_Value>;
08: const auto Calculate = state<class Calculate>;
09:
10: //actions:
11: auto insert_value = [](auto e) { do_something_1() };
12: auto play_animation = [](auto e) { do_something_2() };
13: auto stop_animation = [](auto e) { do_something_3() };
14:
15: struct wait_for_values {
16:   auto operator()() const {
17:     using namespace sml;
18:     //transactions table:
19:     return make_transition_table (
20:      *Wait_For_First_Value + event<value> / insert_value ->
Wait_For_Second_Value,
21:       Wait_For_Second_Value + event<value> / insert_value ->
Calculate,
22:       Calculate + sml::on_entry<_> / play_animation(),
23:       Calculate + sml::on_exit<_> / stop_animation(),
24:       Calculate + event<done> -> Wait_For_First_Value
25:     );
26:   }
27: };
28:
29: int main() {
30:   using namespace sml;
31:
32:   sm<wait_for_values> sm;
33:   static_assert(1 == sizeof(sm), "sizeof(sm) != 1b");
34:   assert(sm.is(Wait_For_First_Value));
35:
36:   sm.process_event(value{});
37:   assert(sm.is(Wait_For_Second_Value));
38:
39:   sm.process_event(value{});
40:   assert(sm.is(Calculate));
41:
42:   sm.process_event(done{});
43:   assert(sm.is(Wait_For_First_Value));
44: }
```

Listing 9

Vergleich der Ansätze

	Handgeschriebene Konstruktionen	Boost::SML
Komplexität	Abhängig von der Anzahl States und Events	Konstant
Erweiterbarkeit/Wartbarkeit	Abhängig von der Anzahl States und Events	Konstant
Testbarkeit	Komplex, fragile Tests	Einfacher
Run time	Abhängig von Code-Qualität, meistens langsamer	Schnell, Jump Table
Compile time	Schnell	Etwas langsamer
Abhängigkeiten	keine	C++14

Die handgeschriebenen Konstruktionen sind meistens komplex. Für die Implementierung der Transitionen brauchen wir zusätzliche Variablen mit Zustandsinformationen und Bedingungen für die Transitions-Logik. Die if-else Konstruktionen sind für Compiler schwierig zu optimieren und brauchen mehrere CPU Takte für die Abarbeitung. Handgeschriebene Zustandsautomaten sind fragil und man muss alle Änderungen vorsichtig durchführen.

In Boost::SLM Framework wird die Transitions-Logik als Jump Tabelle zur Compile-Zeit generiert, das macht den Compile-Process etwas langsamer, reduziert aber die Laufzeit, weil der Compiler den Zustandsautomaten besser optimieren kann. Zusätzlicher Vorteil ist das einfachere Testing, weil Transationen, Aktionen und Guards per Design getrennt sind. Es ist einfach, Einzelteile separat zu testen. Tests sind stabiler, weil bei Änderungen der Zustandsautomatenlogik nur die Tests, die die Transitionen testen, angepasst werden müssen.

Mit Nutzung von Frameworks wie z.B. Boost::SML können wir Entwicklungszeit sparen. Unsere Software wird einfacher zu lesen, zu erweitern und zu testen. Meistens wird die Laufzeit auch schneller, weil der größte Teil von Entscheidungen zu Kompilierzeit getroffen wird.

References

[1] http://www.omg.org/spec/UML/2.5/
[2] http://en.cppreference.com/w/cpp/concept/Callable
[3] http://boost-experimental.github.io/sml/index.html

Autor

Paweł Wiśniewski hat mehre Jahre Erfahrung in verschiedenen Bereichen der Softwareentwicklung. Er hat sich u.a. mit dem Design von Software für Mikrocontrollern, FPGA als auch Desktop Anwendungen beschäftigt. Außer Softwaredesign interessieren ihn Softwareentwicklungsprozesse und deren Verbesserung. Er ist ein Fan von Automation und versucht so viel wie möglich dem Rechner zu überlassen.