Implementierung einer Ampelanlage mit dem State Machine Framework von Qt 5.10.1

Marcel Grüßinger, Seminar: IT-Anwendungen, Hochschule Offenburg

***Abstract*—Dass Zustandsautomaten eine bedeutende Rolle in unserer Gesellschaft spielen, sollte durch den Einsatz in den unterschiedlichsten technischen Geräten, die wir Menschen täglich benutzen, klar sein. Dennoch ist dies häufig nicht der Fall.**

Diese Automaten (englisch: (Finite) State Machine) sind grundsätzlich ein Modell mit dem das Zusammenspiel zwischen endlich vielen Zuständen und deren Übergängen, welche durch Eingaben ausgelöst werden, aufgezeigt wird. Da State Machines eine hohe Komplexität aufweisen können, existieren unterschiedliche Darstellungsformen, wie das UML-Zustandsdiagramm, welche unter anderem das Ziel verfolgen, diese leichter verständlich zu machen.

Ebenso sind gegenwärtig für eine Vielzahl von Programmiersprachen verschiedene Frameworks und Bibliotheken verfügbar, die eine eigene, abgetestete Implementierung der State Machine bzw. des State Patterns bereitstellen.

Im Rahmen dieses Papers wird die Funktionalität des State Machine Frameworks von Qt (Verwendung der Version 5.10.1), welches für die Sprache C++ vorhanden ist, genauer beleuchtet. Diese Beschreibung soll durch die Implementierung einer Beispielapplikation, in diesem Fall handelt es sich um eine einfache Ampelanlage, unterstützt werden.

***Index Terms*—Qt State Machine Framework, SCXML, Zustandsautomaten, Zustandsdiagramme**

I. Einleitung

Finite State Machines, wie die Zustandsautomaten in der englischen Sprache bezeichnet werden, sind laut dem offiziellen „IEEE Standard Glossary of Software Engineering Terminology“ [1] ein Modell, welches aus einer endlichen Anzahl von Zuständen und Übergängen (zwischen diesen Zuständen) besteht. Wobei an diese Zustandsübergänge, die durch Eingaben ausgelöst werden, optional Aktionen beigefügt werden können. [2] fügt dieser Definition hinzu, dass das Verhalten eines Zustandsautomaten nicht alleine durch die Eingaben beschrieben werden kann, sondern die Historie der Eingaben von Nöten ist.

Um diese Endlichkeit auch in der Begrifflichkeit zu verdeutlichen, ist es gebräuchlich auch von endlichen Automaten zu sprechen.

Diese Zustandsautomaten sind allerdings nicht nur eine rein theoretische Abstraktion, sondern finden auch in der Praxis unserer heutigen, modernen Gesellschaft eine häufige Anwendung, wie bspw. in der Software diverser Geräte.

Alleine wenn sich vor Auge geführt wird, dass das Wort „Automat“ in diesem Begriff enthalten ist, sollten jedem Menschen sofort gewisse Parallelen aus dem täglichen Umfeld auffallen. Dies ist jedoch vielen Menschen, trotz des Einsatzes in Kaffee- oder Snack-Automaten, nicht bewusst. Dass sich die Arbeitsweise eines solchen Automaten einfach auf die Theorie einer State Machine transferieren lässt, kann anhand eines einfachen Beispiels dargestellt werden.

Eingangs befindet sich der Snack-Automat in dem Zustand „Warte auf Produktauswahl“. Wählt der Kunde ein Produkt durch einen Knopfdruck (Eingabe), wechselt der Automat in den darauffolgenden Zustand „Produkt ausgewählt“. Um nun in den Zustand „Produkt ausgeben“ zu gelangen, wird von dem Kunden erwartet, solange Geld einzuwerfen (Eingabe) bis ausreichend Geld vorhanden ist. Nachdem das Wechselgeld durch die Maschine ausgegeben wurde (hier ist die Ausgabe eine „interne“ Eingabe), geht diese wieder in den initialen Zustand über. Dieser beschriebene Ablauf wiederholt sich nun endlich oft.

Aufgrund des Problems, dass Zustandsmaschinen nicht immer einfach in einem Fließtext beschrieben werden können, stellt dieses Paper das UML-Zustandsdiagramm und die Beschreibungssprache State Chart XML als mögliche Darstellungsformen vor.

Der Hauptbestandteil dieser Arbeit hat allerdings die Absicht das Qt State Machine Framework anhand der Implementierung einer Ampelanlage (Abbildung 1) näher zu beleuchten.

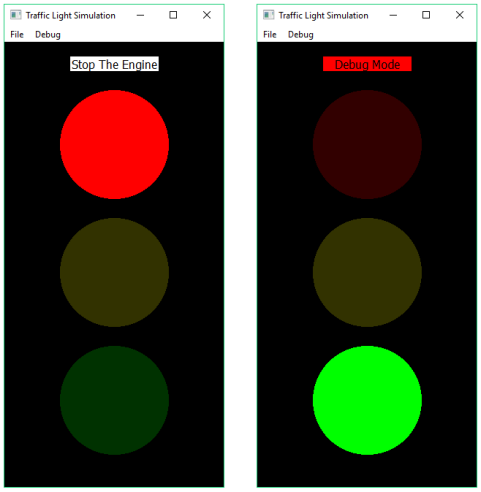

Abbildung 1: Screenshots der Beispielanwendung: Traffic Light Simulation

II. Zustandsautomaten

A. Historie

Alan Turing erbrachte im Jahre 1937 mit der Erfindung der Turingmaschine einen bedeutenden Schritt für die modernen Berechnungsformalismen [3]. Diese Maschinen haben die Eigenschaft unendlich lange Speicherbänder zu besitzen. Dieses Attribut impliziert somit auch, dass eine unbegrenzte Menge an Zuständen existiert. Darum ist die Umsetzung einer Turingmaschine in der Realität, da hier Computer nur über begrenzte Speicherkapazitäten verfügen, nicht immer gegeben.

Die Publikation [3] stellt die Vermutung auf, dass eine Studie von Warren McCulloch und Walter Pitts der Ursprung der endlichen Automaten sein könnte. Diese setzten sich mit der logischen Beschreibung von Vorgängen in Nervennetzen, welche aus einer endlichen Anzahl von Nervenzellen bestehen, auseinander und zogen dabei Parallelen zu der Turingmaschine.

Stephen Cole Kleene griff im Jahre 1956 die Arbeit von McCulloch und Pitts in einem (eigenen) Artikel auf und formulierte die Inhalte zu einem allgemeingültigen Modell zur Darstellung von Ereignissen um. Dieser Artikel wird heute oft in Fachbüchern zitiert, wenn es um den Ursprung von endlichen Automaten geht [3].

Dennoch konnte das Paper [3] die Frage nicht vollständig klären und verweist deshalb in dem Kapitel „Fazit und Ausblick“ auf die nächsten Arbeitsschritte.

B. UML-Zustandsdiagramm

Ein Bild sagt mehr als tausend Worte - diese Auffassung teilt auch David Harel. In dem Paper „Statecharts: A Visual Formalism For Complex Systems“ behandelt Harel eine Erweiterung der vor der Arbeit existierenden Zustandsgraphen. Das Ziel dabei war komplexe Systeme, die meist exponentiell weiterwachsen, in einer graphischen Repräsentation möglichst verständlich darzustellen und zugleich die einfache Skalierbarkeit des Systems zu gestatten [4]. Diese Anpassung bzw. Arbeit besitzt noch heute weitreichende Auswirkungen, da sie die Grundlage für das Zustandsdiagramm der Unified Modeling Language (UML) bildet [5].

Ein Beispiel für solch ein Diagramm ist der Abbildung 2 zu entnehmen.

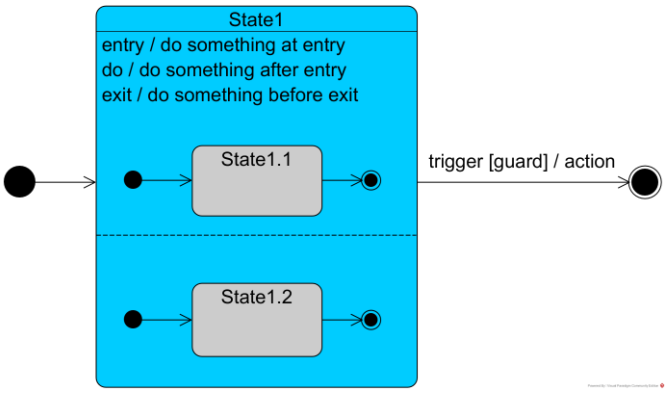

Abbildung 2: Beispiel für ein UML-Zustandsdiagramm

Der Startpunkt wird im Vergleich zu einem Endpunkt mit einem vollständig gefüllten Kreis visualisiert.

Ein Zustand wird mit einem abgerundeten Rechteck, welches durch einen Querstrich in zwei Teilbereiche gegliedert wird, dargestellt. Oberhalb der Linie ist der Name des entsprechenden Zustandes zu finden. Dieser Name sollte zum einen eindeutig und zum anderen kein Verb sein. Im unteren Bereich können maximal zwei Aktionen (entry, exit) und eine Aktivität (do), somit insgesamt drei Verhalten eines Zustandes, spezifiziert werden:

1) entry: wird direkt bei Eintritt in den Zustand ausgeführt
2) do: wird unmittelbar nach der Beendigung der entry-Aktion ausgeführt
3) exit: wird vor dem Verlassen des Zustandes ausgeführt [5]

Im Gegensatz zu Aktivitäten werden Aktionen garantiert vollständig durchlaufen. Es ist ebenfalls möglich, einem Zustand keines der Verhalten zuzuweisen.

Ein Zustand kann neben den Verhalten auch Unterzustände (Kinder) zugeordnet bekommen. Diese werden ebenso in dem Bereich unterhalb der Linie dargestellt werden. Des Weiteren ist es valide diesen Bereich in mehrere Regionen, welche parallel ablaufende Zustände darstellen, zu unterteilen. Diese Regionen werden dabei durch gestrichelte Linien voneinander getrennt [5].

Der Übergang zwischen einem Quell- und einem Zielzustand wird durch einen gerichteten Pfeil visualisiert. Eine Transition muss durch einen Trigger (auch als Event bezeichnet) ausgelöst werden. Dies kann bspw. ein Klick auf einen Button sein. Wurde kein Event für den Zustandsübergang definiert, existieren jedoch Default-Trigger, die nicht explizit umgesetzt werden müssen. Hat der Quellzustand keine Kinder-Zustände, wird er erst dann verlassen, wenn die entry-Aktion und die do-Aktivität ausgeführt wurden. Sind dem Zustand allerdings Kinder untergeordnet, wird der Wechsel erst dann vollzogen, wenn alle Regionen den Endpunkt erreicht haben. Im Englischen wird deshalb in beiden Fällen von einer „completion transition“ gesprochen [5].

Ein Zustandsübergang wird allerdings nur dann ausgelöst, wenn die Guard-Bedingung erfüllt wurde. Ein Beispiel hierfür wäre die Abfrage, ob ausreichend Geld eingeworfen wurde, um das Produkt auszugeben. Ist kein Guard spezifiziert, wird die Transition immer durchgeführt [5].

Weiterhin kann während dem Zustandsübergang noch - nur bei erfolgreicher Prüfung der Bedingung - eine Aktion (nicht zu verwechseln mit dem zuvor genannten Begriff; im Englischen deshalb „effect“) ausgeführt werden, wenn der Übergang stattfindet [5].

C. State Chart XML

Die Beschreibungssprache State Chart XML (SCXML) wurde von dem World Wide Web Consortium (W3C) spezifiziert und veröffentlicht. Diese Sprache erlaubt das Darstellen von Zustandsdiagrammen durch XML-Dateien. SCXML greift dabei Aspekte von CCXML und den von David Harel konzipierten State Charts auf [6].

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
version="1.0 initial="State1">
     <state id="State1">
        <transition target="Final1"/>
        <onentry>
             <log expr="'Hello world'" />
        </onentry>
     </state>
     <final id="Final1"></final>
</scxml/>
```

Dies ist ein einfaches Beispiel eines Zustandsautomaten, der nach dem Eintritt in „State1“ die Phrase „Hello World“ ausgibt und anschließend in den Endzustand „Final1“ wechselt.

SCXML unterstützt unter anderem die Definition von Triggern oder Guard-Bedingungen. Darüber hinaus können eigene Aktionen und Aktivitäten implementiert werden.

III. Qt State Machine Framework

Das State Machine Framework von Qt, in dieser Publikation findet die Version 5.10.1 Verwendung, bietet einige Klassen zur Erstellung und Ausführung von Zustandsautomaten. Es greift dabei die Konzepte und Notationen, die von David Harel in [4] vorgestellt wurden, auf.

Das Framework erlaubt es, die Zustände der endlichen Automaten hierarchisch und/oder parallel anzuordnen (diese Arbeit beschäftigt sich lediglich mit der hierarchischen Verschachtelung) [7]. Das Qt-Event-System und das Signal-Slot-Konzept finden im Rahmen dieses Frameworks Verwendung, um Zustandsübergänge durchzuführen [10]. Die weiteren Funktionalitäten werden innerhalb dieses Kapitels bzw. durch die Beispielanwendung genauer beschrieben.

A. *Qt Framework*

Qt (ausgesprochen „cute“) ist ein plattformübergreifendes Framework, das für Softwareprojekte in der Sprache C++ eingesetzt werden kann. Es wurde 1995 erstmals veröffentlicht. Aktuell ist die „The Qt Company“ für die Wartung und Weiterentwicklung des Frameworks verantwortlich [8].

Das Framework wird in zwei Lizenzierungsmodellen angeboten, nämlich „Open Source“ und „Commercial“. Diese unterscheiden sich hauptsächlich in dem Support und der Lizenz, unter der die programmierte Software verfügbar gemacht werden muss.

Qt enthält neben dem State Machine Framework viele weitere Bibliotheken und APIs, wie zum Beispiel zur Erstellung von graphischen Benutzeroberflächen. Außerdem ist eine eigene IDE, der Qt Creator, in dem Paket inkludiert [9].

B. *Beispielanwendung: Ampelanlage*

Die Beispielanwendung trägt den Namen „Traffic Light Simulation“. Sie besteht aus zwei Schichten: der Präsentations- (Frontend) und der Anwendungsschicht (Backend).

Das Frontend (Abbildung 1) setzt sich dabei aus drei Lampen (in den Farben rot, gelb und grün) und einem Label, dass den aktuellen Zustand der Anlage näher beschreibt, zusammen. Die Ampel kann in dem Simulations-Modus und dem Debug-Modus betrieben werden. Ein Wechsel zwischen diesen Modi wird durch die Betätigung des entsprechenden Buttons in der Menüleiste (unter dem Reiter „Debug“) ausgelöst. Befindet bzw. gelangt die Anlage in den Debug-Modus wird die Simulation pausiert und erst dann wieder fortgesetzt, wenn dieser Zustand verlassen wurde.

Die Geschäftslogik (Backend) der Applikation wird komplett durch einen hierarchisch aufgebauten endlichen Automaten, welcher in Abbildung 3 durch einen Zustandsgraphen visualisiert wurde, abgebildet. Die Hierarchie wurde hierbei in drei Ebenen aufgeteilt:

1) Blaue Ebene: beschreibt die Übergänge zwischen den zwei existierenden Modi
2) Graue Ebene: zeigt die eigentlichen Phasen bzw. Zustände der Ampel auf sowie die Bedingungen, um zwischen diesen zu wechseln
3) Weiße Ebene: beleuchtet den internen Timer, der dafür verantwortlich ist, das Signal für den Übergang in die nächste Ampelphase zu liefern

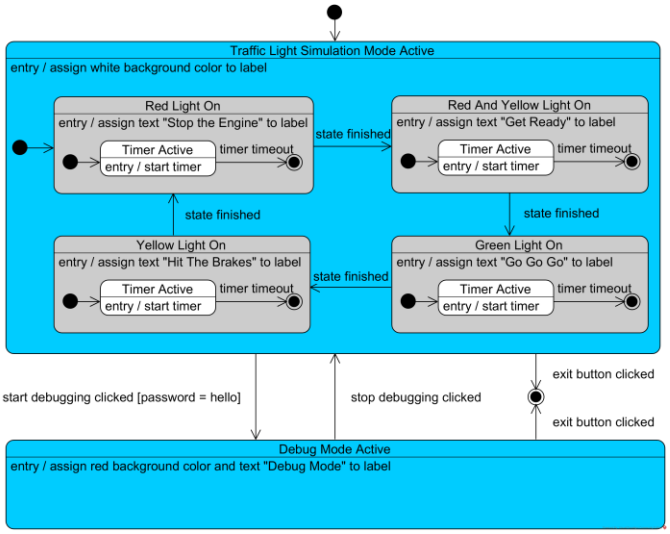

Abbildung 3: Zustandsdiagramm der State Machine der Beispielanwendung

1) *Implementierung der Zustandsmaschine*

In den folgenden Absätzen wird die Implementierung der Zustandsmaschine (Backend) schrittweise vorgestellt. Zur Verbesserung der Verständlichkeit wurde der Quelltext in mehrere Abschnitte aufgeteilt. Die Nummerierung innerhalb der Codeausschnitte gibt Aufschluss über die Reihenfolge der Aufrufe innerhalb der Anwendung.

```
TrafficLight::TrafficLight()
{
    // Erzeugung des Zustandsautomaten
    QStateMachine machine = new QStateMachine();

    // Erzeugung der Modus-Zustaende
    QState* trafficLightSimMode = new QState();
    QState* debugMode = new QState();
    QHistoryState* trafficLightSimModeHistory = new
    QHistoryState(trafficLightSimMode);

     // Erzeugung der Ampelphasen-Zustaende
    QState* redLightOn = new
    QState(trafficLightSimMode);
    QState* innerTimerActive = new
    QState(redLightOn);
    QFinalState* innerTimerDone = new
    QFinalState(redLightOn);

    // (Ende Abschnitt 1)
}
```

Zu Beginn des Konstruktors der TrafficLight-Klasse werden die benötigten Objekte für die State Machine und die jeweiligen Zustände erzeugt. Hierbei wurden ausschließlich die Default-Klassen (QStateMachine, QState, QFinalState, QHistoryState), die von dem Framework angeboten werden, verwendet. Der Konstruktor erhält eigentlich ein Objekt der Klasse „TrafficLightView“ als Übergabeparameter. Dies wurde allerdings aus den Codebeispielen entfernt, um die Komplexität zu reduzieren.

Durch die Übergabe eines QState in den Konstruktor eines anderen Zustandes ist es möglich, eine Hierarchie aufzubauen. Die Ampelanlage setzt dies bspw. bei den Zustandsobjekten „redLightOn“ und „trafficLightSimMode“ ein. In diesem Fall ist der erstere Zustand ein Kind von dem zweiten. Es existiert dabei keine Limitierung bezüglich der Tiefe der Verschachtelung oder der Anzahl von Kindern unterhalb eines Vater-Zustandes.

Die Endzustände eines Automaten können innerhalb des State Machine Frameworks nicht über die QState-Klasse abgebildet werden, sondern ausschließlich über die QFinalState-Klasse. Die Erreichung eines solchen „finalen“ Zustandes löst ein Event aus ([10]), das noch im Rahmen der Aktionen, die einem State zugewiesen werden können, näher beleuchtet wird.

Der QHistoryState stellt eine weitere spezielle Form eines Zustandes dar, denn ein Objekt dieser Klasse speichert bei Unterbrechung den inneren Zustand (also des zuletzt ausgeführten Kindes) des Vaters, welcher dem Konstruktor übergeben wurde. Bei dieser Klasse handelt es sich um einen Pseudo-Zustand. Dies bedeutet, dass solch ein Zustand nicht betreten werden kann. Ein Übergang zu einem QHistoryState wird somit automatisch an das zuletzt ausgeführte Kind weitergeleitet [10]. Bezogen auf die Beispielapplikation kann nun abgeleitet werden, dass wenn die grüne Lampe leuchtet und die Ampel in den Debug-Modus wechselt, eine Rückkehr in den Simulations-Modus dazu führen würde, dass die Ampelanlage wieder in den Zustand der grün leuchtenden Lampe übergeht. Ohne die Verwendung der QHistoryState-Klasse würde in diesem Fall wieder in den initialen Zustand „redLightOn“ zurückgekehrt werden.

```
TrafficLight::TrafficLight()
{
    // Konfig. des Uebergangs zu TrafficLightSimMode
    debugMode->addTransition(stopDebugAction,
    SIGNAL(triggered(bool)),
    trafficLightSimModeHistory);

    // Konfig. des Uebergangs zu DebugMode
    PasswordTransition* transition = new
    PasswordTransition("secret");
    transition->setTargetState(debugMode);
    trafficLightSimMode->addTransition(transition);

    // Konfig. des Uebergangs von rot zu rot-gelb
    redLightOn->addTransition(redLightOn,
    SIGNAL(finished()), redYellowLightOn);

    // (Ende Abschnitt 2)
}
```

Die Transitionen zwischen den Zuständen können innerhalb des Frameworks auf zwei verschiedene Wege spezifiziert werden. Einerseits kann die Methode „addTransition()“, welche sich an dem Signal-Slot-Konzept von Qt bedient, verwendet werden. Diese erwartet als Parameter:

1) ein QObject, welches als Signal-Sender fungiert
2) ein Signal bzw. eine Signal-Methode innerhalb des Senderobjekts
3) einen Zielzustand [10]

Ein Beispiel aus der Ampelanlage wäre der Klick auf das QAction-Element „stopDebugAction“, welcher einen Wechsel von dem Simulations-Modus in den Debug-Modus zur Folge hätte. Ist der Übergang aus dem Vater-Zustand abhängig von der Beendigung der Kinder kann ebenso das Signal „finished()“, welches von der QFinalState-Klasse ausgelöst wird, nach dem identischen Prinzip verwendet werden (vergleiche den Übergang von Rot zu Rot-Gelb).

Alternativ gibt es die Möglichkeit das Event-System, das von Qt zur Verfügung gestellt wird, zu benutzen. Allerdings ist in diesem Fall der Aufwand etwas höher, da hier eine eigene Transition- sowie Event-Klasse implementiert werden muss [10].

```
class PasswordTransition :public QAbstractTransition
{
    Q_OBJECT

    public:
        PasswordTransition(const QString& password);

    protected:
        virtual bool eventTest(QEvent* event);
        virtual void onTransition(QEvent* event);

    private:
        QString password;
};
```

Durch das Erben von der QAbstractTransition-Klasse müssen die Methoden „eventTest()“ und „onTransition()“ innerhalb der Transition-Klasse überschrieben werden. „eventTest()“ ist hauptsächlich dafür verantwortlich die Guard-Bedingungen zu prüfen. Außerdem kann kontrolliert werden, ob das richtige QEvent die Überprüfung ausgelöst hat. Liefert die zuvor beschriebene Methode „true“ zurück, wird „onTransition()“ aufgerufen. In dieser Methode können somit während eines Übergangs weitere Aktionen ausgeführt werden.

Wird in der Anwendung bspw. ein falsches Passwort eingegeben, findet der Wechsel in den Debug-Modus nicht statt.

```
class PasswordEvent : public QEvent
{
    public:
        PasswordEvent(const QString& password) :
        QEvent(QEvent::Type(QEvent::User+1))
        {
            this->password = password;
        }
        QString password;

};
```

Bei der Implementierung der Event-Klasse, die von QEvent erbt, muss vor allem ein Event aus der Enumeration „Type“ (von der Klasse „QEvent“) registriert werden, auf welches gewartet bzw. reagiert werden soll [10]. Dies wird in der

PasswordEvent-Klasse durch den Aufruf des Konstruktors von QEvent erreicht. Hier wird an die erste freie Position des User-Bereichs das Passwort-Event angehängt.

```
QString text = getInputText();
machine->postEvent(new PasswordEvent(text));
```

Damit ein Zustandsübergang von dem Simulations-Modus in den Debug-Modus durch das Event-System durchgeführt werden kann, muss das Event durch den Aufruf der „postEvent()“-Methode mit einem Objekt der QStateMachine-Klasse ausgelöst werden.

```
TrafficLight::TrafficLight()
{
    // Konfig. der Entry-Aktion von innerTimerActive
    connect(innerTimerActive, SIGNAL(entered()),
    timer, SLOT(start()));

    // Konfig. der Entry-Aktion von debugMode
    debugMode ->assignProperty(label,"styleSheet",
    "background-color:rgb(255,0,0);");
    debugMode->assignProperty(label, "text",
    "Debug Mode");

    // (Ende Abschnitt 3)
}
```

Um bei dem Eintritt in einen Zustand eine entry-Aktion auszuführen, kann das Signal-Slot-Konzept verwendet werden. Diese Vorgehensweise ist ebenso auf den Austritt aus einem Zustand transferierbar (Ausführung einer exit-Aktion).

Ein Beispiel für das Betreten liefert Code-Abschnitt 3. Wird in den Zustand „innerTimerActive“ gewechselt, wird durch das Signal „entered()“ für das Timer-Objekt die start()-Methode aufgerufen und somit der Timer gestartet. Soll bei dem Verlassen eines Zustandes eine Aktion durchgeführt werden, ist stattdessen die Verwendung des Signals „exited()“ möglich.

Für entry-Aktionen kann alternativ auch (ohne das Signal-Slot-Konzept) die Methode „assignProperty()“ mit einem Zustandsobjekt aufgerufen werden [10]. Hier ist es bspw. möglich eine konkrete Eigenschaft eines QObjects, wie ein QLabel, anzupassen.

Eine weitere Möglichkeit besteht darin, eine eigene Zustandsklasse zu implementieren und diese von QState erben zu lassen. Durch die Vererbung müssen die Methoden „onEntry()“ und „onExit()“ überschrieben werden.

```
TrafficLight::TrafficLight()
{
    // Konfig. des Startzustands von SimMode
    trafficLightSimMode->setInitialState(
    redLightOn);

    // Konfig. der State Machine
    machine->addState(trafficLightSimMode);
    machine->addState(debugMode);
    machine->setInitialState(trafficLightSimMode);
    machine->start();

    // Setzen einer Animation
    machine->addDefaultAnimation(new
    QPropertyAnimation(label, "styleSheet"));

    // (Ende Abschnitt 4)
}
```

Nach der erfolgreichen Initialisierung der Objekte, demnach bereits direkt nach Code-Abschnitt 1, können die Zustände der Zustandsmaschine mit der Methode „addState()“ hinzugefügt werden. Die Reihenfolge spielt hierbei keine Rolle, da diese durch die (spätere) Spezifikation der Übergänge vorgenommen wird.

Das Setzen des initialen Zustandes für den Automaten muss unter allen Umständen vor dem Aufruf von „start()“ erfolgen, da diesem der Startpunkt sonst unbekannt ist und somit nicht startet.

In dieser Beispielapplikation arbeitet die State Machine Hand in Hand mit einer graphischen Benutzeroberfläche. Das Framework bietet für solche Fälle die Verwendung von Animationen an. Mit diesen können die Zustandsübergänge weicher visualisiert werden, denn statt direkt den Zielwert der Property zuzuweisen, wird sich langsam selbigem angenähert [10]. Im Fall der Ampelanlage betrifft dies bspw. die Property „styleSheet“ (genauer die Hintergrundfarbe).

Um sicherzustellen, dass durch die Animationen kein unerwünschtes Verhalten auftritt, sollte der Zustand erst dann verlassen werden, wenn der Wert der Property tatsächlich gesetzt ist. Dieses Problem kann durch das Signal-Slot-Konzept folgendermaßen gelöst werden:

```
debugMode->addTransition(debugMode,
SIGNAL(propertiesAssigned()), trafficLightSimMode);
```

Es besteht die Möglichkeit eine Animation auf der einen Seite für die gesamte Zustandsmaschine zu setzen. Somit gilt diese für jeden Übergang innerhalb des Automaten. Allerdings besteht auf der anderen Seite die Option, eine Animation nur für eine bestimmte Transition zu verwenden. Diese überschreiben auch global definierte Animationen [10].

2) Implementierung des Frontends

Die Implementierung des Frontends setzt die von Qt verfügbaren GUI-Klassen (QMainWindow, QLabel, etc.) ein. Das Fenster besteht neben dem der Menüleiste und einem Label lediglich aus einer weiteren graphischen Komponente, der „GraphicsView“. Diese beinhaltet eine GraphicsScene, welche quasi eine Leinwand darstellt.

```
void TrafficLightView::drawTrafficLight
(TrafficLightColor* color)
{
     QBrush redBrush(*(color->getRedColor()),
     Qt::SolidPattern);
     QBrush yellowBrush(*(color->getYellowColor()),
     Qt::SolidPattern);
     QBrush greenBrush(*(color->getGreenColor()),
     Qt::SolidPattern);
     QPen pen(Qt::black);

     scene->clear();
     scene->addEllipse(40, 25, 150, 150, pen,
     redBrush);
     scene->addEllipse(40, 200, 150, 150, pen,
     yellowBrush);
     scene->addEllipse(40, 375, 150, 150, pen,
     greenBrush);
}
```

Sie wird durch die Methode „drawTrafficLight()“ zunächst geleert und daraufhin mit drei Ellipsen, welche die einzelnen

Lampen der Ampelanlage visualisieren, befüllt. Die Farben für die Lampen werden durch den Parameter (ein Objekt der Wrapper-Klasse „TrafficLightColor“) geliefert.

IV. FAZIT

Abschließend lässt sich feststellen, dass das Qt State Machine Framework eine einfache und zugleich mächtige Möglichkeit bietet, einen Zustandsautomaten in C++ zu implementieren. Denn ein Softwareentwickler kann durch geringen Aufwand ein (komplexes) System mit verschiedenen Zuständen und deren Übergängen abbilden. Außerdem bietet das Framework durch Polymorphie eine gute Option, eigene Funktionen wie Events oder Transitions, in die bestehende Architektur zu integrieren.

Ausblickend wäre das „Declarative State Machine Framework“, welches ebenso in Qt inkludiert ist, für eine weitere Untersuchung aufzuführen. Dieses erlaubt das Erstellen von Zustandsautomaten mit der Beschreibungssprache QML, welche syntaktisch ähnlich zu SCXML ist. Somit sind nahezu keine Kenntnisse der Programmiersprache C++ erforderlich, um bspw. die vorgestellte Ampelanlage zu implementieren.

REFERENZEN

[1] *“IEEE Standard Glossary of Software Engineering Terminology,” IEEE Std 610.12 1990, pp. 1–84, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=159342, 1990.*

[2] F. Wagner und P. Wolstenholme, „Misunderstandings about state machines“, *Computing and Control Engineering*, Jg. 15, Nr. 4, S. 40–45, 2004.

[3] Z. Lakovic, P. Mai, C. Schenk, W. Ye und Z. Yue, *Über die Entstehung der endlichen Automaten: oder "Warum haben endliche Automaten keinen Namen?"*. [Online] Verfügbar unter: https://files.ifi.uzh.ch/cl/siclemat/lehre/papers/LakovicMai1999.pdf. Zugriff am: Apr. 03 2018.

[4] D. Harel, „STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS“ in *Science of Computer Programming*, S. 231–274.

[5] R. Findenig, T. Leitner, M. Veiten und W. Ecker, „Fast and accurate UML State Chart modeling using TLM$^+$ control flow abstraction“ in *IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010: Anaheim, California, USA, 10 - 12 June 2010*, Anaheim, FL, USA, 2010, S. 97–102.

[6] R. S. Moura und L. A. Guedes, „Simulation of Industrial Applications using the Execution Environment SCXML“ in *5th IEEE International Conference on Industrial Informatics, 2007: 23 - 27 July 2007, Vienna, Austria*, Vienna, Austria, 2007, S. 255–260.

[7] C. Yu *et al.*, „The implementation of IEC60870-5-104 based on UML statechart and Qt state machine framework“ in *2015 5th IEEE International Conference on Electronics Information and Emergency Communication (ICEIEC): 14 - 16 May 2015, Beijing*, Beijing, China, 2015, S. 392–397.

[8] *About Us.* [Online] Verfügbar unter: https://www1.qt.io/company/. Zugriff am: Apr. 07 2018.

[9] The Qt Company, *Get Qt.* [Online] Verfügbar unter: https://www.qt.io/download. Zugriff am: Apr. 07 2018.

[10] The Qt Company, *The State Machine Framework.* [Online] Verfügbar unter: http://doc.qt.io/qt-5/statemachine-api.html. Zugriff am: Apr. 07 2018.